

Clojure

目錄

Clojure入门教程

> 原文：Clojure – Functional Programming for the JVM
>
> 译者：xumingming
>
> 来源：Clojure入门教程

简介

这篇文章的目的是以通俗易懂的方式引导大家进入Clojure的世界。文章涵盖了cojure的大量的特性， 对每一个特性的介绍我力求简介。你不用一条一条往下看，尽管跳到你感兴趣的条目。

请把你的意见，建议发送到mark@ociweb.com(如果是对文章翻译的建议，请直接在文章下面留言: http://xumingming.sinaapp.com/302/clojure-tutorial/)。我对下面这样的建议特别感兴趣:

- 你说是X, 其实是Y
- 你说是X, 但其实说Y会更贴切
- 你没有提到X, 但是我认为X是一个非常重要的话题

对这篇文章的更新可以在 http://www.ociweb.com/mark/clojure/ 找到， 同时你也可以在 http://www.ociweb.com/mark/stm/ 找到有关Software Transactional Memory的介绍， 以及Clojure对STM的实现。

这篇文章里面的代码示例里面通常会以注释的形式说明每行代码的结果/输出，看下面的例子：

```
(+ 1 2) ; showing return value: 3
(println "Hello") ; return nil, showing output:Hello
```

函数式编程

函数式编程 是一种强调函数必须被当成第一等公民对待， 并且这些函数是“纯”的编程方式。这是受 lambda表达式 启发的。纯函数的意思是同一个函数对于同样同样的参数，它的返回值始终是一样的 — 而不会因为前一次调用修改了某个全局变量而使得后面的调用和前面调用的结果不一样。这使得这种程序十分容易理解、调试、测试。它们没有副作用 — 修改某些全局变量， 进行一些IO操作（文件IO和数据库）。状态被维护在方法的参数上面， 而这些参数被存放在栈(stack)上面（通常通过递归调用）， 而不是被维护在全局的堆（heap）上面。这使得方面可以被执行多次而不用担心它会更改什么全局的状态（这是非常重要的特征，等我们讨论事务的时候你就会意识到了）。这也使得高级编译器为了提高代码性能而对代码进行重排(reording)和并行化(parallelizing)成为可能。（并行化代码现在还很少见）

在实际生活中，我们的程序是需要一定的副作用的。Haskel的主力开发Simon Peyton-Jones曾经曰过：

“到最后，任何程序都需要修改状态，一个没有副作用的程序对我们来说只是一个黑盒， 你唯一可以感觉到的是：这个黑盒在变热。。”（ http://oscon.blip.tv/file/324976 ）

问题的关键是我们要控制副作用的范围， 清晰地定位它们，避免这种副作用在代码里面到处都是。

把函数当作“第一公民”的语言可以把函数赋值给一个变量，作为参数来调用别的函数， 同时一个函数也可以返回一个函数。可以把函数作为返回值的能力使得我们选择之后程序的行为。接受函数作为参数的函数我们称为“高阶函数”。从某个方面来说，高阶函数的行为是由传进来的函数来配置的，这个函数可以被执行任意次，也可以从不执行。

函数式语言里面的数据是不可修改的。这使得多个线程可以在不用锁的情况下并发地访问这个数据。因为数据不会改变，所以根本不需要上锁。随着多核处理器的越发流行，函数式语言对并发语言的简化可能是它最大的优点。如果所有这些听起来对你来说很有吸引力而且你准备来学学函数式语言，那么你要有点心理准备。许多人觉得函数式语言并不比面向对象的语言难，它们只是风格不同罢了。而花些时间学了函数式语言之后可以得到上面说到的那些好处，我想还是值得的。比较流行的函数式语言有: Clojure , Common Lisp , Erlang , F# , Haskell , ML) , OCaml , Scheme) , Scala . Clojure和Scala是Java Virtual Machine (JVM)上的语言. 还有一些其它基于JVM的语言: Armed Bear Common Lisp (ABCL) , OCaml-Java and Kawa (Scheme).

Clojure概述

Clojure是一个动态类型的，运行在JVM(JDK5.0以上），并且可以和java代码互操作的函数式语言。这个语言的主要目标之一是使得编写一个有多个线程并发访问数据的程序变得简单。

Clojure的发音和单词closure是一样的。Clojure之父是这样解释Clojure名字来历的

“我想把这就几个元素包含在里面： C (C#), L (Lisp) and J (Java). 所以我想到了Clojure, 而且从这个名字还能想到closure;它的域名又没有被占用;而且对于搜索引擎来说也是个很不错的关键词，所以就有了它了.”

很快Clojure就会移植到.NET平台上了. ClojureCLR是一个运行在Microsoft的CLR的Clojure实现. 在我写这个入门教程的时候ClojureCLR已经处于alpha阶段了.

在2011年7月, ClojureScript项目开始了，这个项目把Clojure代码编译成Javascript代码：看这里 https://github.com/clojure/clojurescript .

Clojure是一个开源语言， licence: Eclipse Public License v 1.0 (EPL). This is a very liberal license. 关于EPL的更多信息看这里: http://www.eclipse.org/legal/eplfaq.php .

运行在JVM上面使得Clojure代码具有可移植性，稳定性，可靠的性能以及安全性。同时也使得我们的Clojure代码可以访问丰富的已经存在的java类库：文件 I/O, 多线程, 数据库操作, GUI编程, web应用等等等等.

Clojure里面的每个操作被实现成以下三种形式的一种: 函数(function), 宏(macro)或者special form. 几乎所有的函数和宏都是用Clojure代码实现的，它们的主要区别我们会在后面解释。Special forms不是用clojure代码实现的，而且被clojure的编译器识别出来. special forms的个数是很少的， 而且现在也不能再实现新的special forms了. 它们包括: catch , def , do , dot (‘.’), finally , fn , if , let , loop , monitor-enter , monitor-exit , new , quote , recur , set! , throw , try 和 var .

Clojure提供了很多函数来操作序列（sequence), 而序列是集合的逻辑视图。很多东西可以被看作序列：Java集合, Clojure的集合, 字符串, 流, 文件系统结构以及XML树. 从已经存在的clojure集合来创建新的集合的效率是非常高的，因为这里使用了persistent data structures 的技术(这对于clojure在数据不可更改的情况下，同时要保持代码的高效率是非常重要的)。

Clojure提供三种方法来安全地共享可修改的数据。所有三种方法的实现方式都是持有一个可以开遍的引用指向一个不可改变的数据。Refs通过使用 Software Transactional Memory （STM）来提供对于多块共享数据的同步访问。Atoms提供对于单个共享数据的同步访问。Agents提供对于单个共享数据的异步访问。这个我们会在 “引用类型”一节详细讨论。

Clojure是 Lisp) 的一个方言. 但是Clojure对于传统的Lisp有所发展。比如, 传统Lisp使用 `car` 来获取链表里面的第一个数据。而Clojure使用 `first。有关更多Clojure和Lisp的不同看这里：` http://clojure.org/lisps .

Lisp的语法很多人很喜欢，很多人很讨厌, 主要因为它大量的使用圆括号以及前置表达式. 如果你不喜欢这些，那么你要考虑一下是不是要学习Clojure了 。许多文件编辑器以及IDE会高亮显示匹配的圆括号, 所以你不用担心需要去人肉数有没有多加一个左括号，少写一个右括号. 同时Clojure的代码还要比java代码简洁. 一个典型的java方法调用是这样的:

```
methodName(arg1, arg2, arg3);
```

而Clojure的方法调用是这样的:

```
(function-name arg1 arg2 arg3)
```

左括号被移到了最前面；逗号和分号不需要了. 我们称这种语法叫：“form”. 这种风格是简单而又美丽：Lisp里面所有东西都是这种风格的.要注意的是clojure里面的命名规范是小写单词，如果是多个单词，那么通过中横线连接。

定义函数也比java里面简洁。Clojure里面的 `println` 会在它的每个参数之间加一个空格。如果这个不是你想要的，那么你可以把参数传给 `str` ，然后再传给 `println` .

```
// Java
public void hello(String name) {
    System.out.println("Hello, " + name);
}
```

```
; Clojure
(defn hello [name]
  (println "Hello," name))
```

Clojure里面大量之用了延迟计算. 这使得只有在我们需要函数结果的时候才去调用它. “懒惰序列” 是一种集合，我们之后在需要的时候才会计算这个集合理解面的元素. 只使得创建无限集合非常高效.

对Clojure代码的处理分为三个阶段：读入期，编译期以及运行期。在读入期，读入期会读取clojure源代码并且把代码转变成数据结构，基本上来说就是一个包含列表的列表的列表。。。。在编译期，这些数据结构被转化成java的bytecode。在运行期这些java bytecode被执行。函数只有在运行期才会执行。而宏在编译期就被展开成实际对应的代码了。

Clojure代码很难理解么？想想每次你看到java代码里面那些复杂语法比如: `if` , `for` , 以及匿名内部类, 你需要停一下来想想它们到底是什么意思（不是那么的直观），同时如果想要做一个高效的Java工程师，我们有一些工具可以利用来使得我

们的代码更容易理解。同样的道理，Clojure也有类似的工具使得我们可以更高效的读懂clojure代码。比如： `let` ， `apply` ， `map` ， `filter` ， `reduce` 以及匿名函数 ... 所有这些我们会在后面介绍.

开始吧

Clojure是一个相对来说很新的语言。在经过一些年的努力之后，Clojure的第一版是在2007年10月16日发布的。Clojure的主要部分被称为 “Clojure proper” 或者 “core”。你可以从这里下载： http://clojure.org/downloads . 你也可以使用 Leiningen 。最新的源代码可以从它的Git库下载.

“Clojure Contrib“是一个大家共享的类库列表。其中有些类库是成熟的，被广泛使用的并且最终可能会被加入Clojure Proper的。但是也有些库不是很成熟，没有被广泛使用，所以也就不会被包含在Conjure Proper里面。所以Clojure Proper里面是鱼龙混杂，使用的时候要自己斟酌，文档在这里： http://richhickey.github.com/clojure-contrib/index.html

对于一个Clojure Contrib， 有三种方法可以得到对应的jar包. 首先你可以下载一个打包好的jar包。其次你可以用maven 来自己打个jar包. Maven可以从这里下载 http://maven.apache.org/ . 打包命令是 “ `mvn package` “. 再其次你可以用ant. ant 可以从这里下载 http://ant.apache.org/ 。命令是： “ `ant -Dclojure.jar={path}` “.

要从最小的源代码来编译clojure, 我们假设你已经安装了 Git 和 Ant , 运行下面的命令来下载并且编译打包Clojure Proper和Clojure Contrib:

```
git clone git://github.com/richhickey/clojure.git
cd clojure
ant clean jar
cd ..
git clone git://github.com/richhickey/clojure-contrib.git
cd clojure-contrib
ant -Dclojure.jar=../clojure/clojure.jar clean jar
```

下一步，写一个脚本来运行Read/Eval/Print Loop (REPL) 以及运行 Clojure 程序. 这个脚本通常被命名为”clj”. 怎么使用REPL我们等会再介绍. Windows下面，最简单的clj脚本是这样的(UNIX, Linux以及 Mac OS X下面把 %1 改成 $1):

```
java -jar /path/clojure.jar %1
```

这个脚本假定 `java` 在你的 `PATH` 环境变量里面. 为了让这个脚本更加有用:

- 把经常使用的JAR包比如 “Clojure Contrib” 以及数据库driver添加到classpath里面去(`-cp`).
- 使clj更好用：用 rlwrap (利用keystrokes来支持的) 或者 JLine 来得到命令提示以及命令历史提示。
- 添加一个启动脚本来设置一些特殊变量(比如 `*print-length*和` `*print-level*`), 加载一些常用的、不再 `java.lang` 里面的包 加载一些常用的不再 `clojure.core` 里面的函数并且定义一些常用自定义的函数.

使用这个脚本来启动REPL我们会等会介绍. 用下面这个命令来运行一个clojure脚本（通常以clj为后缀名）：

```
clj source-file-path
```

更多细节看这里 http://clojure.org/getting_started 以及这里：http://clojure.org/repl_and_main 。同时Stephen Gilardi 还提供了一个脚本：http://github.com/richhickey/clojure-contrib/raw/master/launchers/bash/clj-env-dir 。

为了更充分的利用机器的多核，你应该这样来调用：“ `java -server ...` “.

提供给Clojure的命令行参数被封装在预定义的变量 `*command-line-args*`里面。

Clojure 语法

Lisp方言有一个非常简洁的语法 — 有些人觉得很美的语法。数据和代码的表达形式是一样的，一个列表的列表很自然地在内存里面表达成一个tree。(a b c)表示一个对函数a的调用，而参数是b和c。如果要表示数据，你需要使用 `'(a b c)` o或者 `(quote (a b c))` 。通常情况下就是这样了，除了一些特殊情况 — 到底有多少特殊情况取决于你所使用的方言。

我们把这些特殊情况称为语法糖。语法糖越多代码写起来越简介，但是同时我们也要学习更多的东西以读懂这些代码。这需要找到一个平衡点。很多语法糖都有对应的函数可以调用。到底语法糖是多了还是少了还是你们自己来判断吧。

下面这个表格简要地列举了Clojure里面的一些语法糖， 这些语法糖我们会在后面详细讲解的，所以如果你现在没办法完全理解它们不用担心。

作用	语法糖
注释	`; _text_` 单行注释
字符 (Java `char` 类型)	`\_char_` `\tab` `\newline` `\space` `\u_unicode-hex-valu`
字符串 (Java `String` 对象)	`"_text_"`
关键字是一个内部字符串; 两个同样的关键字指向同一个对象; 通常被用来作为map的key	`:_name_`
当前命名空间的关键字	`::_name_`
正则表达式	`#"_pattern_"`
逗号被当成空白（通常用	

在集合里面用来提高代码可读性）	`,` (逗号）
链表(linked list)	`'(_items_)` (不会evaluate每个元素）
vector（和数组类似）	`[_items_]`
set	`#{_items_}` 建立一个hash-set
map	`{_key-value-pairs_}` 建立一个hash-map
给symbol或者集合绑定元数据	`#^{_key-value-pairs_} _object_` 在读入期处理
获取symbol或者集合的元数据	`^_object_`
获取一个函数的参数列表（个数不定的）	`& _name_`
函数的不需要的参数的默认名字	`_` (下划线)
创建一个java对象（注意class-name后面的点）	`(_class-name_. _args_)`
调用java方法	`(. _class-or-instance_ _method-name_ _args_)` 或者 `(._method-name_ _class-or-instance_ _args_)`
串起来调用多个函	

数，前面一个函数的返回值会作为后面一个函数的第一个参数；你还可以在括号里面指定额外参数；注意前面的两个点	`(.. _class-or-object_ (_method1 args_) (_method2 args_`
创建一个匿名函数	`#(_single-expression_)` 用 `%` (等同于 `%1`), `%1` , `%2来表`
获取Ref, Atom 和 Agent对应的valuea	`@_ref_`
get `Var` object instead of the value of a symbol (var-quote)	`#'_name_`
syntax quote (使用在宏里面)	```
unquote (使用在宏里面)	`~_value_`
unquote splicing (使用在宏里面)	`~@_value_`
auto-gensym (在宏里面用来产生唯一的	`_prefix_#`

symbol名字)	

对于二元操作符比如 +和*, Lisp方言使用前置表达式而不是中止表达式，这和一般的语言是不一样的。比如在java里面你可能会写 `a + b + c` , 而在Lisp里面它相当于
`(+ a b c)` 。这种表达方式的一个好处是如果操作数有多个，那么操作符只用写一次 .
其它语言里面的二元操作符在lisp里面是函数，所以可以有多个操作数。

Lisp代码比其它语言的代码有更多的小括号的一个原因是Lisp里面不使用其它语言使用的大括号，比如在java里面，方法代码是被包含在大括号里面的，而在lisp代码里面是包含在小括号里面的。

比较下面两段简单的Java和Clojure代码，它们实现相同的功能。它们的输出都是：“edray” 和 “orangeay”.

```
// This is Java code.
public class PigLatin {

  public static String pigLatin(String word) {
  char firstLetter = word.charAt(0);
  if ("aeiou".indexOf(firstLetter) != -1) return word + "ay";
  return word.substring(1) + firstLetter + "ay";
  }

  public static void main(String args[]) {
  System.out.println(pigLatin("red"));
  System.out.println(pigLatin("orange"));
  }
}
```

```
; This is Clojure code.
; When a set is used as a function, it returns a boolean
; that indicates whether the argument is in the set.
(def vowel? (set "aeiou"))

(defn pig-latin [word] ; defines a function
  ; word is expected to be a string
  ; which can be treated like a sequence of characters.
  (let [first-letter (first word)] ; assigns a local binding
    (if (vowel? first-letter)
      (str word "ay") ; then part of if
      (str (subs word 1) first-letter "ay")))) ; else part of if

(println (pig-latin "red"))
(println (pig-latin "orange"))
```

Clojure支持所有的常见数据类型比如 booleans (`true` and `false`), 数字, 高精度浮点数, 字符(上面表格里面提到过) 以及字符串. 同时还支持分数 — 不是浮点数，因此在计算的过程中不会损失精度.

Symbols是用来给东西命名的. 这些名字是被限制在名字空间里面的，要么是指定的名字空间，要么是当前的名字空间. Symbols的值是它所代表的名字的值. 要使用Symbol的值，你必须把它用引号引起来.

关键字以冒号打头，被用来当作唯一标示符，通常用在map里面 (比如 `:red` , `:green` 和 `:blue`).

和任何语言一样，你可以写出很难懂的Clojure代码。遵循一些最佳实践可以避免这个。写一些简短的，专注自己功能的函数可以使函数变得容易读，测试以及重复利用。经常使用“抽取方法”的模式来对你的代码进行重构。高度内嵌的函数是非常难懂得，千万不要这么写， 你可以使用let来帮助你。把匿名函数传递给命名函数是非常常见的，但是不要把一个匿名函数传递给另外一个匿名函数， 这样代码就很难懂了。

REPL

REPL 是read-eval-print loop的缩写. 这是Lisp的方言提供给用户的一个标准交互方式，如果用过python的人应该用过这个，你输入一个表达式，它立马再给你输出结果，你再输入。。。如此循环。这是一个非常有用的学习语言，测试一些特性的工具。

为了启动REPL， 运行我们上面写好的clj脚本。成功的话会显示一个” `user=>` “.“ `=>` ”前面的字符串表示当前的默认名字空间。“=>”后面的则是你输入的form以及它的输出结果。 下面是个简单的例子：

```
user=> (def n 2)
#'user/n
user=> (* n 3)
6
```

`def` 是一个 special form， 它相当于java里面的定义加赋值语句. 它的输出表示一个名字叫“ `n` ”的symbol被定义在当前的名字空间“ `user` ”里面。

要查看一个函数，宏或者名字空间的文档输入 `(doc _name_)`。看下面的例子：

```
(require 'clojure.contrib.str-utils)
(doc clojure.contrib.str-utils/str-join) ; ->
; -------------------------
; clojure.contrib.str-utils/str-join
; ([separator sequence])
;   Returns a string of all elements in 'sequence', separated by
;   'separator'.  Like Perl's 'join'.
```

如果要找所有包含某个字符串的所有的函数的，宏的文档，那么输入这个命令 `(find-doc "_text_")` .

如果要查看一个函数，宏的源代码 `(source _name_)` . `source` 是一个定义在 `clojure.contrib.repl-utils` 名字空间里面的宏，REPL会自动加载这个宏的。

如果要加载并且执行文件里面的clojure代码那么使用这个命令 `(load-file "_file-path_")` . Clojure源文件一般以.clj作为后缀。

如果要退出REPL，在Windows下面输出ctrl-z然后回车， 或者直接 ctrl-c; 在其它平台下 (包括UNIX, Linux 和 Mac OS X), 输入 ctrl-d.

变量

Clojure里面是不支持变量的。它跟变量有点像，但是在被赋值之前是不允许改的，包括：全局binding, 线程本地(thread local)binding， 以及函数内的本地binding，以及一个表达式内部的binding。

`def` 这个special form 定义一个全局的 binding，并且你还可以给它一个”root value” ，这个root value在所有的线程里面都是可见的，除非你给它赋了一个线程本地的值. `def` 也可以用来改变一个已经存在的binding的root value —— 但是这是不被鼓励的，因为这会牺牲不可变数据所带来的好处。

函数的参数是只在这个函数内可见的本地binding。

`let` 这个special form 创建局限于一个 当前form的bindings. 它的第一个参数是一个vector, 里面包含名字-表达式的对子。表达式的值会被解析然后赋给左边的名字。这些binding可以在这个vector后面的表达式里面使用。这些binding还可以被多次赋值以改变它们的值，let命令剩下的参数是一些利用这个binding来进行计算的一些表达式。注意：如果这些表达式里面有调用别的函数，那么这个函数是无法利用let创建的这个binding的。

宏 `binding` 跟 `let 类似` , 但是它创建的本地binding会暂时地覆盖已经存在的全局binding. 这个binding可以在创建这个binding的form以及这个form里面调用的函数里面都能看到。但是一旦跳出了这个 `binding` 那么被覆盖的全局binding的值会回复到之前的状态。

从 Clojure 1.3开始, binding只能用在 动态变量(dynamic var)上面了. 下面的例子演示了怎么定一个dynamic var。另一个区别是 `let` 是串行的赋值的, 所以后面的binding可以用前面binding的值, 而 `binding` 是不行的.

要被用来定义成新的、本地线程的、用binding来定义的binding有它们自己的命名方式：她们以星号开始，以星号结束。在这篇文章里面你会看到： `*command-line-args*` , `*agent*` , `*err*` , `*flush-on-newline*` , `*in*` , `*load-tests*` , `*ns*` , `*out*` , `*print-length*` , `*print-level*` and `*stack-trace-depth*` .要使用这些binding的函数会被这些binding的值影响的。比如给*out*一个新的binding会改变println函数的输出终端。

下面的例子介绍了 `def` , `let` 和 `binding` 的用法。

```
(def ^:dynamic v 1) ; v is a global binding

(defn f1 []
  (println "f1: v =" v)) ; global binding

(defn f2 []
  (println "f2: before let v =" v) ; global binding
  (let [v 2] ; creates local binding v that shadows global one
    (println "f2: in let, v =" v) ; local binding
    (f1))
  (println "f2: after let v =" v)) ; global binding

(defn f3 []
  (println "f3: before binding v =" v) ; global binding
  (binding [v 3] ; same global binding with new, temporary value
    (println "f3: in binding, v =" v) ; global binding
    (f1))
  (println "f3: after binding v =" v)) ; global binding

(defn f4 []
 (def v 4)) ; changes the value of the global binding

(f2)
(f3)
(f4)
(println "after calling f4, v =" v)
```

上面代码的输出是这样的：

```
f2: before let v = 1
f2: in let, v = 2
f1: v = 1 (let DID NOT change value of global binding)
f2: after let v = 1
f3: before binding v = 1
f3: in binding, v = 3
f1: v = 3 (binding DID change value of global binding)
f3: after binding v = 1 (value of global binding reverted back)
after calling f4, v = 4
```

集合

Clojure提供这些集合类型: list, vector, set, map。同时Clojure还可以使用Java里面提供的将所有的集合类型，但是通常不会这样做的， 因为Clojure自带的集合类型更适合函数式编程。

Clojure集合有着java集合所不具备的一些特性。所有的clojure集合是不可修改的、异源的以及持久的。不可修改的意味着一旦一个集合产生之后，你不能从集合里面删除一个元素，也往集合里面添加一个元素。异源的意味着一个集合里面可以装进任何东西（而不必须要这些东西的类型一样）。持久的以为着当一个集合新的版本产生之后，旧的版本还是在的。CLojure以一种非常高效的，共享内存的方式来实现这个的。比如有一个map里面有一千个name-valuea pair, 现在要往map里面加一个，那么对于那些没有变化的元素， 新的map会共享旧的map的内存，而只需要添加一个新的元素所占用的内存。

有很多核心的函数可以用来操作所有这些类型的集合。。多得以至于无法在这里全部描述。其中的一小部分我们会在下面介绍vector的时候介绍一下。要记住的是，因为clojure里面的集合是不可修改的，所以也就没有对集合进行修改的函数。相反clojure里面提供了一些函数来从一个已有的集合来高效地创建新的集合 — 使用 persistent data structures 。同时也有一些函数操作一个已有的集合（比如vector)来产生另外一种类型的集合(比如LazySeq), 这些函数有不同的特性。

提醒： 这一节里面介绍的Clojure集合对于学习clojure来说是非常的重要。但是这里介绍一个函数接着一个函数，所以你如果觉得有点烦，有点乏味，你可以跳过，等用到的时候再回过头来查询。

`count` 返回集合里面的元素个数，比如：

```
(count [19 "yellow" true]) ; -> 3
```

`conj` 函数是 conjoin的缩写, 添加一个元素到集合里面去，到底添加到什么位置那就取决于具体的集合了，我们会在下面介绍具体集合的时候再讲。

`reverse` 把集合里面的元素反转。

```
(reverse [2 4 7]) ; -> (7 4 2)
```

`map` 对一个给定的集合里面的每一个元素调用一个指定的方法，然后这些方法的所有返回值构成一个新的集合（LazySeq）返回。这个指定了函数也可以有多个参数，那么你就需要给map多个集合了。如果这些给的集合的个数不一样，那么执行这个函数的次数取决于个数最少的集合的长度。比如：

```
; The next line uses an anonymous function that adds 3 to its argum
(map #(+ % 3) [2 4 7]) ; -> (5 7 10)
(map + [2 4 7] [5 6] [1 2 3 4]) ; adds corresponding items -> (8 12
```

`apply` 把给定的集合里面的所有元素一次性地给指定的函数作为参数调用，然后返回这个函数的返回值。所以apply与map的区别就是map返回的还是一个集合，而apply返回的是一个元素， 可以把apply看作是SQL里面的聚合函数。比如：

```
(apply + [2 4 7]); -> 13
```

有很多函数从一个集合里面获取一个元素，比如：

```
(def stooges ["Moe" "Larry" "Curly" "Shemp"])
(first stooges) ; -> "Moe"
(second stooges) ; -> "Larry"
(last stooges) ; -> "Shemp"
(nth stooges 2) ; indexes start at 0 -> "Curly"
```

也有一些函数从一个集合里面获取多个元素，比如：

```
(next stooges) ; -> ("Larry" "Curly" "Shemp")
(butlast stooges) ; -> ("Moe" "Larry" "Curly")
(drop-last 2 stooges) ; -> ("Moe" "Larry")
; Get names containing more than three characters.
(filter #(> (count %) 3) stooges) ; -> ("Larry" "Curly" "Shemp")
(nthnext stooges 2) ; -> ("Curly" "Shemp")
```

有一些谓词函数测试集合里面每一个元素然后返回一个布尔值，这些函数都是"short-circuit"的，一旦它们的返回值能确定它们就不再继续测试剩下的元素了，有点像java的&&和or, 比如:

```
(every? #(instance? String %) stooges) ; -> true
(not-every? #(instance? String %) stooges) ; -> false
(some #(instance? Number %) stooges) ; -> nil
(not-any? #(instance? Number %) stooges) ; -> true
```

列表

Lists是一个有序的元素的集合 — 相当于java里面的LinkedList。这种集合对于那种一直要往最前面加一个元素，干掉最前面一个元素是非常高效的(O(1)) — 想到于java里面的堆栈, 但是没有高效的方法来获取第N个元素， 也没有高效的办法来修改第N个元素。

下面是创建同样的list的多种不同的方法：

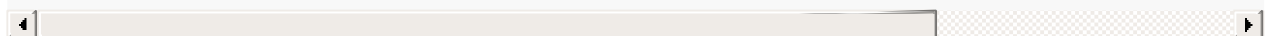
```
(def stooges (list "Moe" "Larry" "Curly"))
(def stooges (quote ("Moe" "Larry" "Curly")))
(def stooges '("Moe" "Larry" "Curly"))
```

`some` 可以用来检测一个集合是否含有某个元素. 它的参数包括一个谓词函数以及一个集合。你可以能会想了，为了要看一个list到底有没有某个元素为什么要指定一个谓词函数呢？其实我们是故意这么做来让你尽量不要这么用的。从一个list里面搜索一个元素是线性的操作（不高效），而要从一个set里面搜索一个元素就容易也高效多了，看下面的例子对比：

```
(some #(= % "Moe") stooges) ; -> true
(some #(= % "Mark") stooges) ; -> nil
; Another approach is to create a set from the list
; and then use the contains? function on the set as follows.
(contains? (set stooges) "Moe") ; -> true
```

`conj` 和 `cons` 函数的作用都是通过一个已有的集合来创建一个新的包含更多元素的集合 — 新加的元素在最前面。 `remove` 函数创建一个只包含所指定的谓词函数测试结果为false的元素的集合:

```
(def more-stooges (conj stooges "Shemp")) -> ("Shemp" "Moe" "Larry"
(def less-stooges (remove #(= % "Curly") more-stooges)) ; -> ("Shen
```



`into` 函数把两个list里面的元素合并成一个新的大list

```
(def kids-of-mike '("Greg" "Peter" "Bobby"))
(def kids-of-carol '("Marcia" "Jan" "Cindy"))
(def brady-bunch (into kids-of-mike kids-of-carol))
(println brady-bunch) ; -> (Cindy Jan Marcia Greg Peter Bobby)
```

`peek` 和 `pop` 可以用来把list当作一个堆栈来操作. 她们操作的都是list的第一个元素。

向量

Vectors也是一种有序的集合。这种集合对于从最后面删除一个元素，或者获取最后面一个元素是非常高效的(O(1))。这意味着对于向vector里面添加元素使用conj被使用cons更高效。Vector对于以索引的方式访问某个元素（用nth命令）或者修改某个元素(用assoc)来说非常的高效。函数定义的时候指定参数列表用的就是vector。

下面是两种创建vector的方法：

```
(def stooges (vector "Moe" "Larry" "Curly"))
(def stooges ["Moe" "Larry" "Curly"])
```

除非你要写的程序要特别用到list的从前面添加/删除效率很高的这个特性， 否则一般来说我们鼓励你们用vector而不是lists。这主要是因为语法上 `[...]` 比‘ `(...)` 更自然，更不容易弄混淆。因为函数，宏以及special form的语法也是 `(...)`。

`get` 获取vector里面指定索引的元素. 我们后面会看到get也可以从map里面获取指定key的value。索引是从0开始的。 `get` 函数和函数 `nth` 类似. 它们都接收一个可选的默认值参数 — 如果给定的索引超出边界，那么会返回这个默认值。如果没有指定默认值而索引又超出边界了， `get` 函数会返回 `nil` 而 `nth` 会抛出一个异常. 看例子:

```
(get stooges 1 "unknown") ; -> "Larry"
(get stooges 3 "unknown") ; -> "unknown"
```

`assoc` 可以对 vectors 和 maps进行操作。 当用在 vector上的时候, 它会从给定的vector创建一个新的vector, 而指定的那个索引所对应的元素被替换掉。如果指定的这个索引等于vector里面元素的数目，那么我们会把这个元素加到新vector的最后面去；如果指定的索引比vector的大小要大，那么一个 `IndexOutOfBoundsException` 异常会被抛出来。看代码:

```
(assoc stooges 2 "Shemp") ; -> ["Moe" "Larry" "Shemp"]
```

`subvec` 获取一个给定vector的子vector。它接受三个参数，一个vectore, 一个起始索引以及一个可选的结束索引。如果结束索引没有指定，那么默认的结束索引就是vector的大小。新的vector和原来的vector共享内存(所以高效)。

所有上面的对于list的例子代码对于vector同样适用。 `peek` 和 `pop` 函数对于vector同样适用, 只是它们操作的是vector的最后一个元素，而对于list操作的则是第一个函数。 `conj` 函数从一个给定的vector创建一个新的vector — 添加一个元素到新的vector的最后面去. `cons` 函数从一个给定的vector创建一个新的vector — 添加一个新的元素到vector的最前面去。

集合

Sets 是一个包含不重复元素的集合。当我们要求集合里面的元素不可以重复，并且我们不要求集合里面的元素保持它们添加时候的顺序，那么sets是比较适合的。Clojure 支持两种不同的set： 排序的和不排序的。如果添加到set里面的元素相互之间不能比较大小，那么一个 `ClassCastException` 异常会被抛出来。下面是一些创建set的方法:

```
(def stooges (hash-set "Moe" "Larry" "Curly")) ; not sorted
(def stooges #{"Moe" "Larry" "Curly"}) ; same as previous
(def stooges (sorted-set "Moe" "Larry" "Curly"))
```

`contains?` 函数可以操作sets和maps. 当操作set的时候, 它返回给定的set是否包含某个元素。这比在list和vector上面使用的 `some函数就简单多了` . 看例子:

```
(contains? stooges "Moe") ; -> true
(contains? stooges "Mark") ; -> false
```

Sets 可以被当作它里面的元素的函数来使用. 当以这种方式来用的时候，返回值要么是这个元素，要么是nil. 这个比起contains？函数来说更简洁. 比如:

```
(stooges "Moe") ; -> "Moe"
(stooges "Mark") ; -> nil
(println (if (stooges person) "stooge" "regular person"))
```

在介绍list的时候提到的函数 `conj` 和 `into` 对于set也同样适用. 只是元素的顺序只有对sorted-set才有定义.

`disj` 函数通过去掉给定的set里面的一些元素来创建一个新的set. 看例子:

```
(def more-stooges (conj stooges "Shemp")) ; -> #{"Moe" "Larry" "Cu
(def less-stooges (disj more-stooges "Curly")) ; -> #{"Moe" "Larry"
```

你也可以看看 `clojure.set` 名字空间里面的一些函数: `difference` , `index` , `intersection` , `join` , `map-invert` , `project` , `rename` , `rename-keys` , `select` 和 `union` . 其中有些函数的操作的对象是map而不是set。

映射

Maps 保存从key到value的a对应关系 — key和value都可以是任意对象。key-value组合被以一种可以按照key的顺序高效获取的方式保存着。

下面是创建map的一些方法， 其中逗号是为了提高可读性的，它是可选的，解析的时候会被当作空格忽略掉的。

```
(def popsicle-map
  (hash-map :red :cherry, :green :apple, :purple :grape))
(def popsicle-map
  {:red :cherry, :green :apple, :purple :grape}) ; same as previous
(def popsicle-map
  (sorted-map :red :cherry, :green :apple, :purple :grape))
```

Map可以作为它的key的函数，同时如果key是keyword的话，那么key也可以作为map的函数。下面是三种获取:green所对应的值的方法：

```
(get popsicle-map :green)
(popsicle-map :green)
(:green popsicle-map)
```

`contains?` 方法可以操作 sets 和 maps. 当被用在map上的时候，它返回map是否包含给定的key. `keys` 函数返回map里面的所有的key的集合. `vals` 函数返回map里面所有值的集合. 看例子:

```
(contains? popsicle-map :green) ; -> true
(keys popsicle-map) ; -> (:red :green :purple)
(vals popsicle-map) ; -> (:cherry :apple :grape)
```

`assoc` 函数可以操作 maps 和 vectors. 当被用在map上的时候，它会创建一个新的map， 同时添加任意对新的name-value pair, 如果某个给定的key已经存在了，那么它的值会被更新。看例子:

```
(assoc popsicle-map :green :lime :blue :blueberry)
; -> {:blue :blueberry, :green :lime, :purple :grape, :red :cherry}
```

`dissoc` 创建一个新的map， 同时忽略掉给定的那么些key， 看例子:

```
(dissoc popsicle-map :green :blue) ; -> {:purple :grape, :red :cher
```

我们也可以把map看成一个简单的集合，集合里面的每个元素是一个pair: name-value: `clojure.lang.MapEntry` 对象. 这样就可以和doseq跟destructuring一起使用了, 它们的作用都是更简单地来遍历map， 我们会在后面详细地介绍这些函数. 下面的这个例子会遍历 `popsicle-map` 里面的所有元素，把key bind到 `color`， 把value bind到 `flavor`。 `name`函数返回一个`keyword`的字符串名字。

```
(doseq [[color flavor] popsicle-map]
  (println (str "The flavor of " (name color)
    " popsicles is " (name flavor) ".")))
```

上面的代码的输出是这样的：

```
The flavor of green popsicles is apple.
The flavor of purple popsicles is grape.
The flavor of red popsicles is cherry.
```

`select-keys` 函数接收一个map对象，以及一个key的集合的参数，它返回这个集合里面key在那个集合里面的一个子map。看例子:

```
(select-keys popsicle-map [:red :green :blue]) ; -> {:green :apple,
```

`conj` 函数添加一个map里面的所有元素到另外一个map里面去。如果目标map里面的key在源map里面也有，那么目标map的值会被更新成源map里面的值。

map里面的值也可以是一个map， 而且这样嵌套无限层。获取嵌套的值是非常简单的。同样的，更新一个嵌套的值也是很简单的。

为了证明这个， 我们会创建一个描述人（person）的map。其中内嵌了一个表示人的地址的map，同时还有一个叫做employer的内嵌map。

```
(def person {
  :name "Mark Volkmann"
  :address {
    :street "644 Glen Summit"
    :city "St. Charles"
    :state "Missouri"
    :zip 63304}
  :employer {
    :name "Object Computing, Inc."
    :address {
      :street "12140 Woodcrest Executive Drive, Suite 250"
      :city "Creve Coeur"
      :state "Missouri"
      :zip 63141}}})
```

`get-in` 函数、宏 `->` 以及函数 `reduce` 都可以用来获得内嵌的key. 下面展示了三种获取这个人的employer的address的city的值的方法：

```
(get-in person [:employer :address :city])
(-> person :employer :address :city) ; explained below
(reduce get person [:employer :address :city]) ; explained below
```

宏 `->` 我们也称为 “thread” 宏, 它本质上是调用一系列的函数，前一个函数的返回值作为后一个函数的参数. 比如下面两行代码的作用是一样的:

```
(f1 (f2 (f3 x)))
(-> x f3 f2 f1)
```

在名字空间 `clojure.contrib.core` 里面还有个 -?>宏， 它会马上返回nil， 如果它的调用链上的任何一个函数返回nil (short-circuit)。这会避免抛出 `NullPointerException` 异常。

`reduce` 函数接收一个需要两个参数的函数, 一个可选的value以及一个集合。它会以value以及集合的第一个元素作为参数来调用给定的函数（如果指定了value的话），要么以集合的第一个元素以及第二个元素为参数来调用给定的函数（如果没有指定value的话)。接着就以这个返回值以及集合里面的下一个元素为参数来调用给定的函数，知道集合里面的元素都被计算了 — 最后返回一个值. 这个函数与ruby里面的 `inject` 以及Haskell里面的 `foldl` 作用是一样的。

`assoc-in` 函数可以用来修改一个内嵌的key的值。看下面的例子把person的employer->address->city修改成Clayton了。

```
(assoc-in person [:employer :address :city] "Clayton")
```

`update-in` 函数也是用来更新给定的内嵌的key对应的值，只是这个新值是通过一个给定的函数来计算出来。下面的例子里面会把person的employer->address->zip改成旧的zip + “-1234″。看例子:

```
(update-in person [:employer :address :zip] str "-1234") ; using th
```

StructMap

StructMap和普通的map类似，它的作用其实是用来模拟java里面的javabean， 所以它比普通的map的优点就是，它把一些常用的字段抽象到一个map里面去，这样你就不用一遍一遍的重复了。并且和java类似，他会帮你生成合适的 `equals` 和 `hashCode` 方法。并且它还提供方式让你可以创建比普通map里面的hash查找要快的字段访问方法(javabean里面的getXXX方法)。

`create-struct` 函数 和 `defstruct` 宏都可以用来定义StructMap, defstruct内部调用的也是 `create-struct` 。map的key通常都是用keyword来指定的。看例子:

```
(def vehicle-struct (create-struct :make :model :year :color)) ; lo
(defstruct vehicle-struct :make :model :year :color) ; short way
```

`struct` 实例化StructMap的一个对象，相当于java里面的new关键字. 你提供给struct的参数的顺序必须和你定义的时候提供的keyword的顺序一致，后面的参数可以忽略， 如果忽略，那么对应key的值就是nil。看例子:

```
(def vehicle (struct vehicle-struct "Toyota" "Prius" 2009))
```

`accessor` 函数可以创建一个类似java里面的getXXX的方法， 它的好处是可以避免hash查找， 它比普通的hash查找要快。看例子:

```
; Note the use of def instead of defn because accessor returns
; a function that is then bound to "make".
(def make (accessor vehicle-struct :make))
(make vehicle) ; -> "Toyota"
(vehicle :make) ; same but slower
(:make vehicle) ; same but slower
```

在创建一个StructMap之后， 你还可以给它添加在定义struct的时候没有指定的key。但是你不能删除定义时候已经指定的key。

函数定义

`defn` 宏用来定义一个函数。它的参数包括一个函数名字，一个可选的注释字符串，参数列表，然后一个方法体。而函数的返回值则是方法体里面最后一个表达式的值。所有的函数都会返回一个值， 只是有的返回的值是nil。看例子:

```
(defn parting
  "returns a String parting"
  [name]
  (str "Goodbye, " name)) ; concatenation

(println (parting "Mark")) ; -> Goodbye, Mark
```

函数必须先定义再使用。有时候可能做不到， 比如两个方法项目调用，clojure采用了和C语言里面类似的做法： declare, 看例子:

```
(declare <em>function-names</em>)
```

通过宏 `defn-` 定义的函数是私有的. 这意味着它们只在定义它们的名字空间里面可见. 其它一些类似定义私有函数/宏的还有： `defmacro-` 和 `defstruct-` (在 `clojure.contrib.def` 里面)。

函数的参数个数可以是不定的。可选的那些参数必须放在最后面(这一点跟其它语言是一样的), 你可以通过加个&符号把它们收集到一个list里面去Functions can take a variable number of parameters. Optional parameters must appear at the end. They are gathered into a list by adding an ampersand and a name for the list at the end of the parameter list.

```
(defn power [base & exponents]
  ; Using java.lang.Math static method pow.
  (reduce #(Math/pow %1 %2) base exponents))
(power 2 3 4) ; 2 to the 3rd = 8; 8 to the 4th = 4096
```

函数定义可以包含多个参数列表以及对应的方法体。每个参数列表必须包含不同个数的参数。这通常用来给一些参数指定默认值。看例子：

```
(defn parting
  "returns a String parting in a given language"
  ([] (parting "World"))
  ([name] (parting name "en"))
  ([name language]
    ; condp is similar to a case statement in other languages.
    ; It is described in more detail later.
    ; It is used here to take different actions based on whether th
    ; parameter "language" is set to "en", "es" or something else.
    (condp = language
      "en" (str "Goodbye, " name)
      "es" (str "Adios, " name)
      (throw (IllegalArgumentException.
        (str "unsupported language " language))))))

(println (parting)) ; -> Goodbye, World
(println (parting "Mark")) ; -> Goodbye, Mark
(println (parting "Mark" "es")) ; -> Adios, Mark
(println (parting "Mark", "xy"))
; -> java.lang.IllegalArgumentException: unsupported language xy
```

匿名函数是没有名字的。他们通常被当作参数传递给其他有名函数(相对于匿名函数)。匿名函数对于那些只在一个地方使用的函数比较有用。下面是定义匿名函数的两种方法：

```
(def years [1940 1944 1961 1985 1987])
(filter (fn [year] (even? year)) years) ; long way w/ named argumen
(filter #(even? %) years) ; short way where % refers to the argumen
```

通过 `fn` 定义的匿名函数可以包含任意个数的表达式；而通过 `#(...)` ，定义的匿名函数则只能包含一个表达式，如果你想包含多个表达式，那么把它用 `do` 包起来。如果只有一个参数，那么你可以通过 `%` 来引用它；如果有多个参数，那么可以通过 `%1` ，`%2` 等等来引用。看例子:

```
(defn pair-test [test-fn n1 n2]
  (if (test-fn n1 n2) "pass" "fail"))

; Use a test-fn that determines whether
; the sum of its two arguments is an even number.
(println (pair-test #(even? (+ %1 %2)) 3 5)) ; -> pass
```

Java里面的方法可以根据参数的类型来进行重载。而Clojure里面则只能根据参数的个数来进行重载。不过Clojure里面的multimethods技术可以实现任意 类型的重载。

宏 `defmulti` 和 `defmethod` 经常被用在一起来定义 multimethod. 宏 `defmulti` 的参数包括一个方法名以及一个dispatch函数，这个dispatch函数的返回值会被用来选择到底调用哪个重载的函数。宏 `defmethod` 的参数则包括方法名，dispatch的值， 参数列表以及方法体。一个特殊的dispatch值 `:default` 是用来表示默认情况的 — 即如果其它的dispatch值都不匹配的话，那么就调用这个方法。 `defmethod` 多定义的名字一样的方法，它们的参数个数必须一样。传给multimethod的参数会传给dipatch函数的。

下面是一个用multimethod来实现基于参数的类型来进行重载的例子：

```
(defmulti what-am-i class) ; class is the dispatch function
(defmethod what-am-i Number [arg] (println arg "is a Number"))
(defmethod what-am-i String [arg] (println arg "is a String"))
(defmethod what-am-i :default [arg] (println arg "is something else
(what-am-i 19) ; -> 19 is a Number
(what-am-i "Hello") ; -> Hello is a String
(what-am-i true) ; -> true is something else
```

因为dispatch函数可以是任意一个函数，所以你也可以写你自己的dispatch函数。比如一个自定义的dispatch函数可以会根据一个东西的尺寸大小来返回 `:small` , `:medium` 以及 `:large` 。然后对应每种尺寸有一个方法。

下划线可以用来作为参数占位符 ?– 如果你不要使用这个参数的话。这个特性在回调函数里面比较有用， 因为回调函数的设计者通常想把尽可能多的信息给你， 而你通常可能只需要其中的一部分。看例子：

```
(defn callback1 [n1 n2 n3] (+ n1 n2 n3)) ; uses all three arguments
(defn callback2 [n1 _ n3] (+ n1 n3)) ; only uses 1st & 3rd argument
(defn caller [callback value]
  (callback (+ value 1) (+ value 2) (+ value 3)))
(caller callback1 10) ; 11 + 12 + 13 -> 36
(caller callback2 10) ; 11 + 13 -> 24
```

`complement` 函数接受一个函数作为参数，如果这个参数返回值是true， 那么它就返回false, 相当于一个取反的操作。看例子:

```
(defn teenager? [age] (and (>= age 13) (< age 20)))
(def non-teen? (complement teenager?))
(println (non-teen? 47)) ; -> true
```

`comp` 把任意多个函数组合成一个，前面一个函数的返回值作为后面一个函数的参数。 调用的顺序是从右到左（注意不是从左到右） 看例子：

```
(defn times2 [n] (* n 2))
(defn minus3 [n] (- n 3))
; Note the use of def instead of defn because comp returns
; a function that is then bound to "my-composition".
(def my-composition (comp minus3 times2))
(my-composition 4) ; 4*2 - 3 -> 5
```

`partial` 函数创建一个新的函数 — 通过给旧的函数制定一个初始值， 然后再调用原来的函数。比如 `*` 是一个可以接受多个参数的函数，它的作用就是计算它们的乘积，如果我们想要一个新的函数，使的返回结果始终是乘积的2倍，我们可以这样做：

```
; Note the use of def instead of defn because partial returns
; a function that is then bound to "times2".
(def times2 (partial * 2))
(times2 3 4) ; 2 * 3 * 4 -> 24
```

下面是一个使用 `map` 和 `partial` 的有趣的例子.

```
(defn- polynomial
  "computes the value of a polynomial
   with the given coefficients for a given value x"
  [coefs x]
  ; For example, if coefs contains 3 values then exponents is (2 1
  (let [exponents (reverse (range (count coefs)))]
    ; Multiply each coefficient by x raised to the corresponding ex
    ; and sum those results.
    ; coefs go into %1 and exponents go into %2.
    (apply + (map #(* %1 (Math/pow x %2)) coefs exponents))))

(defn- derivative
  "computes the value of the derivative of a polynomial
   with the given coefficients for a given value x"
  [coefs x]
  ; The coefficients of the derivative function are obtained by
  ; multiplying all but the last coefficient by its corresponding e
  ; The extra exponent will be ignored.
  (let [exponents (reverse (range (count coefs)))
        derivative-coefs (map #(* %1 %2) (butlast coefs) exponents)
    (polynomial derivative-coefs x)))

(def f (partial polynomial [2 1 3])) ; 2x^2 + x + 3
(def f-prime (partial derivative [2 1 3])) ; 4x + 1

(println "f(2) =" (f 2)) ; -> 13.0
(println "f'(2) =" (f-prime 2)) ; -> 9.0
```

下面是另外一种做法 (Francesco Strino建议的).

%1 = a, %2 = b, result is ax + b

%1 = ax + b, %2 = c, result is (ax + b)x + c = ax^2 + bx + c

```
(defn- polynomial
  "computes the value of a polynomial
   with the given coefficients for a given value x"
  [coefs x]
  (reduce #(+ (* x %1) %2) coefs))
```

`memoize` 函数接受一个参数，它的作用就是给原来的函数加一个缓存，所以如果同样的参数被调用了两次， 那么它就直接从缓存里面返回缓存了的结果，以提高效率， 但是当然它会需要更多的内存。(其实也只有函数式编程里面能用这个技术，因为函数没有side-effect, 多次调用的结果保证是一样的)

`time` 宏可以看成一个wrapper函数， 它会打印被它包起来的函数的执行时间，并且返回这个函数的返回值。看下面例子里面是怎么用的。

下面的例子演示在多项式的的计算里面使用memoize:

```
; Note the use of def instead of defn because memoize returns
; a function that is then bound to "memo-f".
(def memo-f (memoize f))

(println "priming call")
(time (f 2))

(println "without memoization")
; Note the use of an underscore for the binding that isn't used.
(dotimes [_ 3] (time (f 2)))

(println "with memoization")
(dotimes [_ 3] (time (memo-f 2)))
```

上面代码的输出是这样的：

```
priming call
"Elapsed time: 4.128 msecs"
without memoization
"Elapsed time: 0.172 msecs"
"Elapsed time: 0.365 msecs"
"Elapsed time: 0.19 msecs"
with memoization
"Elapsed time: 0.241 msecs"
"Elapsed time: 0.033 msecs"
"Elapsed time: 0.019 msecs"
```

从上面的输出我们可以看到好几个东西。首先第一个方法调用比其它的都要长很多。– 其实这和用不用memonize没有什么关系。第一个memoize调用所花的时间也要比其他memoize调用花的时间要长， 因为要操作缓存，其它的memoize调用就要快很多了。

Java 互操作

Clojure程序可以使用所有的java类以及接口。和在java里面一样 `java.lang` 这个包里面的类是默认导入的。你可以手动的用 `import` 函数来导入其它包的类。看例子:

```
(import
  '(java.util Calendar GregorianCalendar)
  '(javax.swing JFrame JLabel))
```

同时也可以看下宏ns下面的 `[:import](http://xumingming.sinaapp.com/302/clojure-functional-progr`

有两种方式可以访问类里面的常量的：

```
(. java.util.Calendar APRIL) ; -> 3
(. Calendar APRIL) ; works if the Calendar class was imported
java.util.Calendar/APRIL
Calendar/APRIL ; works if the Calendar class was imported
```

在Clojure代码里面调用java的方法是很简单的。因此很多java里面已经实现的功能Clojure就没有实现自己的了。比如, Clojure里面没有提供函数来计算一个数的绝对值，因为可以用 `java.lang.Math` 里面的abs方法。而另一方面，比如这个类里面还提供了一个 `max` 方法来计算两个数里面比较大的一个, 但是它只接受两个参数，因此Clojure里面自己提供了一个可以接受多个参数的max函数。

有两种方法可以调用java里面的静态方法：

```
(. Math pow 2 4) ; -> 16.0
(Math/pow 2 4)
```

同样也有两种方法来创建一个新的java的对象，看下面的例子。这里注意一下我们用 `def` 创建的对象bind到一个全局的binding。这个其实不是必须的。有好几种方式可以得到一个对象的引用比如把它加入一个集合或者把它传入一个函数。

```
(import '(java.util Calendar GregorianCalendar))
(def calendar (new GregorianCalendar 2008 Calendar/APRIL 16)) ; Apr
(def calendar (GregorianCalendar. 2008 Calendar/APRIL 16))
```

同样也有两种方法可以调用java对象的方法:

```
(. calendar add Calendar/MONTH 2)
(. calendar get Calendar/MONTH) ; -> 5
(.add calendar Calendar/MONTH 2)
(.get calendar Calendar/MONTH) ; -> 7
```

一般来说我们比较推荐使用下面那种用法(.add, .get), 上面那种用法在定义宏的时候用得比较多， 这个等到我们讲到宏的时候再做详细介绍。

方法调用可以用 `..` 宏串起来:

```
(. (. calendar getTimeZone) getDisplayName) ; long way
(.. calendar getTimeZone getDisplayName) ; -> "Central Standard Tir
```

还一个宏： `.?.` 在 `clojure.contrib.core` 名字空间里面， 它和上面..这个宏的区别是，在调用的过程中如果有一个返回结果是nil, 它就不再继续调用了，可以防止出现 `NullPointerException` 异常。

`doto` 函数可以用来调用一个对象上的多个方法。它返回它的第一个参数， 也就是所要调用方法的对象。这对于初始化一个对象的对各属性是非常方便的。 (看下面"Namespaces"那一节的 `JFrame` GUI 对象的例子). 比如:

```
(doto calendar
  (.set Calendar/YEAR 1981)
  (.set Calendar/MONTH Calendar/AUGUST)
  (.set Calendar/DATE 1))
(def formatter (java.text.DateFormat/getDateInstance))
(.format formatter (.getTime calendar)) ; -> "Aug 1, 1981"
```

`memfn` 宏可以自动生成代码以使得java方法可以当成clojure里面的"一等公民"来对待。这个可以用来替代clojure里面的匿名方法。当用 `memfn` 来调用java里面那些需要参数的方法的时候， 你必须给每个参数指定一个名字，以让clojure知道你要调用的方法需要几个参数。这些名字到底是什么不重要，但是它们必须要是唯一的， 因为要用这些名字来生成Clojure代码的。下面的代码用了一个map方法来从第二个集合里面取beginIndex来作为参数调用第一个集合里面的字符串的substring方法。大家可以看一下用匿名函数和用memfn来直接调用java的方法的区别。

```
(println (map #(.substring %1 %2)
           ["Moe" "Larry" "Curly"] [1 2 3])) ; -> (oe rry ly)

(println (map (memfn substring beginIndex)
           ["Moe" "Larry" "Curly"] [1 2 3])) ; -> same
```

代理

`proxy` 创建一个继承了指定类并且/或者实现了0个或者多个接口的类的对象。这对于创建那种必须要实现某个接口才能得到通知的listener对象很有用。举一个例子， 大家可以看下面 “Desktop Applications” 那一节的例子。那里我们创建了一个继承JFrame类并且实现ActionListener接口的类的对象。

线程

所有的Clojure方法都实现了 `[java.lang.Runnable](http://java.sun.com/javase/6/docs/api/java/lang` 接口和 `[java.util.concurrent.Callable](http://java.sun.com/javase/6/docs/ap` 接口。这使得非常容易把Clojure里面函数和java里面的线程一起使用。比如：

```
(defn delayed-print [ms text]
  (Thread/sleep ms)
  (println text))

; Pass an anonymous function that invokes delayed-print
; to the Thread constructor so the delayed-print function
; executes inside the Thread instead of
; while the Thread object is being created.
(.start (Thread. #(delayed-print 1000 ", World!"))) ; prints 2nd
(print "Hello") ; prints 1st
; output is "Hello, World!"
```

异常处理

Clojure代码里面抛出来的异常都是运行时异常。当然从Clojure代码里面调用的java代码还是可能抛出那种需要检查的异常的。 `try` , `catch` , `finally` 以及 `throw` 提供了和java里面类似的功能:

```
(defn collection? [obj]
  (println "obj is a" (class obj))
  ; Clojure collections implement clojure.lang.IPersistentCollecti
  (or (coll? obj) ; Clojure collection?
      (instance? java.util.Collection obj))) ; Java collection?

(defn average [coll]
  (when-not (collection? coll)
    (throw (IllegalArgumentException. "expected a collection")))
  (when (empty? coll)
    (throw (IllegalArgumentException. "collection is empty")))
  ; Apply the + function to all the items in coll,
  ; then divide by the number of items in it.
  (let [sum (apply + coll)]
    (/ sum (count coll))))

(try
  (println "list average =" (average '(2 3))) ; result is a clojure
  (println "vector average =" (average [2 3])) ; same
  (println "set average =" (average #{2 3})) ; same
  (let [al (java.util.ArrayList.)]
    (doto al (.add 2) (.add 3))
    (println "ArrayList average =" (average al))) ; same
  (println "string average =" (average "1 2 3 4")) ; illegal argume
  (catch IllegalArgumentException e
    (println e)
    ;(.printStackTrace e) ; if a stack trace is desired
  )
  (finally
    (println "in finally")))
```

上面代码的输出是这样的：

```
obj is a clojure.lang.PersistentList
list average = 5/2
obj is a clojure.lang.LazilyPersistentVector
vector average = 5/2
obj is a clojure.lang.PersistentHashSet
set average = 5/2
obj is a java.util.ArrayList
ArrayList average = 5/2
obj is a java.lang.String
#<IllegalArgumentException java.lang.IllegalArgumentException:
expected a collection>
in finally
```

条件处理

`if` 这个special form跟java里面的if的语义是一样的， 它接受三个参数， 第一个是需要判断的条件，第二个表达式是条件成立的时候要执行的表达式，第三个参数是可选的，在条件不成立的时候执行。如果需要执行多个表达式，那么把多个表达式包在do里面。看例子：

```
(import '(java.util Calendar GregorianCalendar))
(let [gc (GregorianCalendar.)
      day-of-week (.get gc Calendar/DAY_OF_WEEK)
      is-weekend (or (= day-of-week Calendar/SATURDAY) (= day-of-we
  (if is-weekend
    (println "play")
    (do (println "work")
        (println "sleep"))))
```

宏 `when` 和 `when-not` 提供和if类似的功能， 只是它们只在条件成立（或者不成立）时候执行一个表达式。另一个不同是，你可以执行任意数目的表达式而不用用do把他们包起来。

```
(when is-weekend (println "play"))
(when-not is-weekend (println "work") (println "sleep"))
```

宏 `if-let` 把一个值bind到一个变量，然后根据这个binding的值来决定到底执行哪个表达式。下面的代码会打印队列里面第一个等待的人的名字，或者打印“no waiting”如果队列里面没有人的话。

```
(defn process-next [waiting-line]
  (if-let [name (first waiting-line)]
    (println name "is next")
    (println "no waiting")))

(process-next '("Jeremy" "Amanda" "Tami")) ; -> Jeremy is next
(process-next '()) ; -> no waiting
```

`when-let` 宏跟 `if-let` 类似, 不同之处跟上面 `if` 和 `when` 的不同之处是类似的。 他们没有else部分，同时还支持执行任意多个表达式。比如:

```
(defn summarize
  "prints the first item in a collection
  followed by a period for each remaining item"
  [coll]
  ; Execute the when-let body only if the collection isn't empty.
  (when-let [head (first coll)]
    (print head)
    ; Below, dec subtracts one (decrements) from
    ; the number of items in the collection.
    (dotimes [_ (dec (count coll))] (print \.))
    (println)))

(summarize ["Moe" "Larry" "Curly"]) ; -> Moe..
(summarize []) ; -> no output
```

`condp` 宏跟其他语言里面的switch/case语句差不多。它接受两个参数，一个谓词参数 (通常是 `=` 或者 `instance?`) 以及一个表达式作为第二个参数。在这之后，它接受任意数量的值-表达式的对子，这些对子会按顺序evaluate。如果谓词的条件跟某个值匹配了， 那么对应的表达式就被执行。一个可选的最后一个参数可以指定， 这个参数指定如果一个条件都不符合的话， 那么就返回这个值。如果这个值没有指定，而且没有一个条件符合谓词， 那么一个 `IllegalArgumentException` 异常就会被抛出。

下面的例子让用户输入一个数字，如果用户输入的数字是1，2，3，那么程序会打印这些数字对应的英文单词。否则它会打印"unexpected value"。在那之后，它会测试一个本地binding的类型，如果是个数字它会打印这个数字乘以2的结果；如果是字符串， 那么打印这个字符串的长度乘以2的结果。

```
(print "Enter a number: ") (flush) ; stays in a buffer otherwise
(let [reader (java.io.BufferedReader. *in*) ; stdin
      line (.readLine reader)
      value (try
              (Integer/parseInt line)
              (catch NumberFormatException e line))] ; use string v
  (println
    (condp = value
      1 "one"
      2 "two"
      3 "three"
      (str "unexpected value, \"" value \")))
  (println
    (condp instance? value
      Number (* value 2)
      String (* (count value) 2))))
```

`cond` 宏接受任意个 谓词/结果表达式 的组合。它按照顺序来测试所有的谓词，直到有一个谓词的测试结果是true， 那么它返回其所对应的结果。如果没有一个谓词的测试结果是true， 那么会抛出一个 `IllegalArgumentException` 异常。通常最后一个谓词一般都是true, 以充当默认情况。

下面的例子让用户输入水的温度， 然后打印出水的状态： 是冻住了，还是烧开了，还是一般状态。

```
(print "Enter water temperature in Celsius: ") (flush)
(let [reader (java.io.BufferedReader. *in*)
      line (.readLine reader)
      temperature (try
        (Float/parseFloat line)
        (catch NumberFormatException e line))] ; use string value i
  (println
    (cond
      (instance? String temperature) "invalid temperature"
      (<= temperature 0) "freezing"
      (>= temperature 100) "boiling"
      true "neither")))
```

迭代

有很多方法可以遍历一个集合。

宏 `dotimes` 会执行给定的表达式一定次数, 一个本地binding会被给定值：从0到一个给定的数值. 如果这个本地binding是不需要的 (下面例子里面的 `card-number`), 可以用下划线来代替， 看例子:

```
(dotimes [card-number 3]
  (println "deal card number" (inc card-number))) ; adds one to car
```

注意下上面例子里面的 `inc` 函数是为了让输出变成 1, 2, 3 而不是 0, 1, 2。上面代码的输出是这样的:

```
deal card number 1
deal card number 2
deal card number 3
```

宏 `while` 会一直执行一个表达式只要指定的条件为true. 下面例子里面的 `while` 会一直执行，只要这个线程没有停:

```
(defn my-fn [ms]
  (println "entered my-fn")
  (Thread/sleep ms)
  (println "leaving my-fn"))

(let [thread (Thread. #(my-fn 1))]
  (.start thread)
  (println "started thread")
  (while (.isAlive thread)
    (print ".")
    (flush))
  (println "thread stopped"))
```

上面代码的输出是这样的:

```
started thread
.....entered my-fn.
.............leaving my-fn.
thread stopped
```

列表推导式

宏 `for` 和 `doseq` 可以用来做list comprehension. 它们支持遍历多个集合 (最右边的最快) ，同时还可以做一些过滤用 `:when` 和 `:while`。 宏 `for` 只接受一个表达式 , 它返回一个懒惰集合作为结果. 宏 `doseq` 接受任意数量的表达式, 以有副作用的方式执行它们, 并且返回 `nil` .

下面的例子会打印一个矩阵里面 所有的元素出来。 它们会跳过 “B” 列 并且只输出小于3的那些行。我们会在“序列”那一节介绍 `dorun` , 它会强制提取 for 所返回的懒惰集合.

```
(def cols "ABCD")
(def rows (range 1 4)) ; purposely larger than needed to demonstrat

(println "for demo")
(dorun
  (for [col cols :when (not= col \B)
   row rows :while (< row 3)]
    (println (str col row))))

(println "\ndoseq demo")
(doseq [col cols :when (not= col \B)
   row rows :while (< row 3)]
  (println (str col row)))
```

上面的代码的输出是这样的：

```
for demo
A1
A2
C1
C2
D1
D2

doseq demo
A1
A2
C1
C2
D1
D2
```

宏 `loop` 是一个special form, 从它的名字你就可以猜出来它是用来遍历的. 它以及和它类似的 `recur` 会在下一节介绍.

递归

递归发生在一个函数直接或者间接调用自己的时候。一般来说递归的退出条件有检查一个集合是否为空，或者一个状态变量是否变成了某个特定的值(比如0)。这一种情况一般利用连续调用集合里面的 `next` 函数来实现。后一种情况一般是利用 `dec` 函数来递减某一个变量来实现。

如果递归的层次太深的话，那么可能会产生内存不足的情况。所以一些编程语言利用 " tail call optimization " (TCO)的技术来解决这个问题。但是目前Java和Clojure都不支持这个技术。在Clojure里面避免这个问题的一个办法是使用special form: `loop` 和 `recur` 。另一个方法是使用 trampoline 函数。

`loop` / `recur` 组合把一个看似递归的调用变成一个迭代 — 迭代不需要占用栈空间。 `loop` special form 跟 `let` special form 类似的地方是它们都会建立一个本地binding，但是同时它也建立一个递归点， 而这个递归点就是recur的参数里面的那个函数。 `loop` 给这些binding一个初始值。对 `recur` 的调用使得程序的控制权返回给 `loop` 并且给那些本地binding赋了新的值。给recur传递的参数一定要和loop所创建的binding的个数一样。同样recur只能出现在loop这个special form的最后一行。

```
(defn factorial-1 [number]
  "computes the factorial of a positive integer
   in a way that doesn't consume stack space"
  (loop [n number factorial 1]
    (if (zero? n)
      factorial
      (recur (dec n) (* factorial n)))))

(println (time (factorial-1 5))) ; -> "Elapsed time: 0.071 msecs"\r
```

`defn` 宏跟 `loop` special form一样也会建立一个递归点。 `recur` special form也可以被用在一个函数的最后一句用来把控制权返回到函数的第一句并以新的参数重新执行。

另外一种实现 factorial 函数的方法是使用 `reduce` 函数。这个我们在 "集合" 那一节就已经介绍过了。它支持一种更加"函数"的方式来做这个事情。不过不幸的是，在这种情况下，它的效率要低一点。注意一下 `range` 函数返回一个数字的范围，这个范围包括它的左边界，但是不包括它的右边界。

```
(defn factorial-2 [number] (reduce * (range 2 (inc number))))

(println (time (factorial-2 5))) ; -> "Elapsed time: 0.335 msecs"\r
```

你可以把上面的 `reduce` 换成 `apply,` 可以得到同样的结果，但是apply要更慢一点。这也说明了我们要熟悉每个方法的特点的重要性，以在各个场合使用合适的函数。

`recur` 不支持那种一个函数调用另外一个函数，然后那个函数再回调这个函数的这种递归。但是我们没有提到的 `[trampoline](http://clojure.github.com/clojure/clojure.core-api.html` 函数是支持的。

谓词

Clojure 提供了很多函数来充当谓词的功能 — 测试条件是否成立。它们的返回值是true或者false。在Clojure里面 `false` 以及 `nil` 被解释成false. `true` 以及任何其他值都被解释成true， 包括0。谓词函数的名字一般以问号结尾。

反射是一种获取一个对象的特性，而不是它的值的过程。比如说对象的类型。有很多谓词函数进行反射。 测试一个对象的类型的谓词包括 `class?` , `coll?` , `decimal?` , `delay?` , `float?` , `fn?` , `instance?` , `integer?` , `isa?` , `keyword?` , `list?` , `macro?` , `map?` , `number?` , `seq?` , `set?` , `string?` 以及 `vector?` 。一些非谓词函数也进行反射操作，包括： `ancestors` , `bases` , `class` , `ns-publics` 以及 `parents` 。

测试两个值之间关系的谓词有： `<` , `<=` , `=` , `not=` , `==` , `>` , `>=` , `compare` , `distinct?` 以及 `identical?` .

测试逻辑关系的谓词有： `and` , `or` , `not` , `true?` , `false?` 和 `nil?`

测试集合的一些谓词在前面已经讨论过了，包括： `empty?` , `not-empty` , `every?` , `not-every?` , `some?` 以及 `not-any?` .

测试数字的谓词有 `even?` , `neg?` , `odd?` , `pos?` 以及 `zero?` .

序列

序列可以看成是集合的一个逻辑视图。许多事物可以看成是序列。包括Java的集合，Clojure提供的集合，字符串，流，目录结构以及XML树。

很多Clojure的函数返回一个lazy序列(LazySeq), 这种序列里面的元素不是实际的数据， 而是一些方法， 它们直到用户真正需要数据的时候才会被调用。LazySeq的一个好处是在你创建这个序列的时候你不用太担心这个序列到底会有多少元素。下面是会返回lazySeq的一些函数: `cache-seq` , `concat` , `cycle` , `distinct` , `drop` , `drop-last` , `drop-while` , `filter` , `for` , `interleave` , `interpose` , `iterate` , `lazy-cat` , `lazy-seq` , `line-seq` , `map` , `partition` , `range` , `re-seq` , `remove` , `repeat` , `replicate` , `take` , `take-nth` , `take-while` and `tree-seq` 。

LazySeq是刚接触Clojure的人比较容易弄不清楚的一个东西。比如你们觉得下面这个代码的输出是什么？

```
(map #(println %) [1 2 3])
```

当在一个REPL里面运行的时候，它会输出 1, 2 和 3 在单独的行上面， 以及三个nil(三个println的返回结果)。REPL总是立即解析/调用我们所输入的所有的表达式。但是当作为一个脚本来运行的时候，这句代码不会输出任何东西。因为 `map` 函数返回的是一个LazySeq。

有很多方法可以强制LazySeq对它里面的方法进行调用。比如从序列里面获取一个元素的方法 `first` , `second` , `nth` 以及 `last` 都能达到这个效果。序列里面的方法是按顺序调用的， 所以你如果要获取最后一个元素， 那么整个LazySeq里面的方法都会被调用。

如果LazySeq的头被存在一个binding里面，那么一旦一个元素的方法被调用了， 那么这个元素的值会被缓存起来， 下次我们再来获取这个元素的时候就不用再调用函数了。

`dorun` 和 `doall` 函数迫使一个LazySeq里面的函数被调用。 `doseq` 宏, 我们在 "迭代" 那一节提到过的, 会迫使一个或者多个LazySeq里面的函数调用。 `for` 宏, 也在是"迭代"那一节提到的，不会强制调用LazySeq里面的方法， 相反， 他会返回另外一个LazySeq。

为了只是简单的想要迫使LazySeq里面的方法被调用，那么 `doseq` 或者 `dorun` 就够了。调用的结果不会被保留的， 所以占用的内存也就比较少。这两个方法的返回值都是 `nil` . 如果你想调用的结果被缓存， 那么你应该使用 `doall` .

下面的表格列出来了强制LazySeq里面的方法被调用的几个办法。

结果要缓存	只要求方法被执行，不需要缓存	
操作单个序列	`doall`	`dorun`
利用list comprehension语法来操作多个序列	N/A	`doseq`

一般来说我们比较推荐使用 `doseq` 而不是 `dorun` 函数， 因为这样代码更加易懂。 同时代码效率也更高， 因为dorun内部使用map又创建了另外一个序列。比如下面的两会的结果是一样的。

```
(dorun (map #(println %) [1 2 3]))
(doseq [i [1 2 3]] (println i))
```

如果一个方法会返回一个LazySeq并且在它的方法被调用的时候还会有副作用，那么大多数情况下我们应该使用 `doall` 来调用并且返回它的结果。这使得副作用的出现时间更容易确定。否则的话别的调用者可能会调用这个LazySeq多次，那么副作用也就会出现多次 -- 从而可能出现错误的结果。

下面的几个表达式都会在不同的行输出1, 2, 3, 但是它们的返回值是不一样的。 `do` special form 是用来实现一个匿名函数，这个函数先打印这个值， 然后再把这个值返回。

```
(doseq [item [1 2 3]] (println item)) ; -> nil
(dorun (map #(println %) [1 2 3])) ; -> nil
(doall (map #(do (println %) %) [1 2 3])) ; -> (1 2 3)
```

LazySeq使得创建无限序列成为可能。因为只有需要使用的数据才会在用到的时候被调用创建。比如

```
(defn f
  "square the argument and divide by 2"
  [x]
  (println "calculating f of" x)
  (/ (* x x) 2.0))

; Create an infinite sequence of results from the function f
; for the values 0 through infinity.
; Note that the head of this sequence is being held in the binding
; This will cause the values of all evaluated items to be cached.
(def f-seq (map f (iterate inc 0)))

; Force evaluation of the first item in the infinite sequence, (f (
(println "first is" (first f-seq)) ; -> 0.0

; Force evaluation of the first three items in the infinite sequenc
; Since the (f 0) has already been evaluated,
; only (f 1) and (f 2) will be evaluated.
(doall (take 3 f-seq))

(println (nth f-seq 2)) ; uses cached result -> 2.0
```

下面的代码和上面的代码不一样的地方是，在下面的代码里面LazySeq的头没有被保持在一个binding里面，所以被调用过的方法的返回值不会被缓存。所以它所需要的内存比较少，但是如果同一个元素被请求多次，那么它的效率会低一点。

```
(defn f-seq [] (map f (iterate inc 0)))
(println (first (f-seq))) ; evaluates (f 0), but doesn't cache resu
(println (nth (f-seq) 2)) ; evaluates (f 0), (f 1) and (f 2)
```

另外一种避免保持LazySeq的头的办法是把这个LazySeq直接传给函数：

```
(defn consumer [seq]
  ; Since seq is a local binding, the evaluated items in it
  ; are cached while in this function and then garbage collected.
  (println (first seq)) ; evaluates (f 0)
  (println (nth seq 2))) ; evaluates (f 1) and (f 2)

(consumer (map f (iterate inc 0)))
```

输入/输出

Clojure提供了很少的方法来进行输入/输出的操作。因为我们在Clojure代码里面可以很轻松的使用java里面的I/O操作方法。但是?clojure.java.io 库使得使用java的I/O方法更加简单。

这些预定义的special symbols `*in*` , `*out*` 以及 `*err*` 默认被设定成 stdin, stdout 以及 stderr 。如果要flush `*out*` ,里面的输出，使用 `(flush)` 方法，效果和 `(.flush *out*)` 一样。当然这些symbol的binding是可以改变的。比如你可以把输出重定向到 " `my.log` "文件里面去。 看下面的例子：

```
(binding [*out* (java.io.FileWriter. "my.log")]
  ...
  (println "This goes to the file my.log.")
  ...
  (flush))
```

`print` 可以打印任何对象的字符串表示到*out*，并且在两个对象之间加一个空格。

`println` 函数和 `print` 类似， 但是它会在最后加一个newline符号。默认的话它还会有一个flush的动作。这个默认动作可以通过把 special symbol `*flush-on-newline*` 设成 `false` 来取消掉。

`newline` 函数写一个newline符号 `*out*` 流里面去。 在调用 `print` 函数后面手动调用 `newline` 和直接调用 `println` 的效果是一样的。

`pr` 与 `prn` 是和 `print` 与 `println` 想对应的一对函数, 但是他们输出的形式可以被 Clojure reader去读取。它们对于把Clojure的对象进行序列化的时候比较有用。默认情况下它们不会打印数据的元数据。可以通过把 special symbol `*print-meta*` 设置成 `true` 来调整这个行为。

下面的例子演示了我们提到的四个打印方法。注意使用print和pr输出的字符串的不同之处。

```
(let [obj1 "foo"
      obj2 {:letter \a :number (Math/PI)}] ; a map
  (println "Output from print:")
  (print obj1 obj2)

  (println "Output from println:")
  (println obj1 obj2)

  (println "Output from pr:")
  (pr obj1 obj2)

  (println "Output from prn:")
  (prn obj1 obj2))
```

上面代码的输出是这样的：

```
Output from print:
foo {:letter a, :number 3.141592653589793}Output from println:
foo {:letter a, :number 3.141592653589793}
Output from pr:
"foo" {:letter \a, :number 3.141592653589793}Output from prn:
"foo" {:letter \a, :number 3.141592653589793}
```

所有上面讨论的几个打印函数都会在它们的参数之间加一个空格。你可以通过 `str` 函数来预先组装好要打印的字符串来避免这个行为， 看下面例子：

```
(println "foo" 19) ; -> foo 19
(println (str "foo" 19)) ; -> foo19
```

`print-str` , `println-str` , `pr-str` 以及 `prn-str` 函数 `print` , `println` , `pr` 跟 `prn` 类似, 只是它们返回一个字符串，而不是把他们打印出来。

`printf` 函数和 `print` 类似。但是它接受一个format字符串。 `format` 函数和 `printf` , 类似，只是它是返回一个字符串而不是打印出来。

宏 `with-out-str` 把它的方法体里面的所有输出汇总到一个字符串里面并且返回。

`with-open` 可以自动关闭所关联的连接（.close)方法， 这对于那种像文件啊，数据库连接啊，比较有用，它有点像C#里面的using语句。

`line-seq` 接受一个 `java.io.BufferedReader` 参数，并且返回一个LazySeq, 这个LazySeq包含所有的一行一行由BufferedReader读出的文本。返回一个LazySeq的好处在于，它不用马上读出文件的所有的内容， 这会占用太大的内存。相反， 它只需要在需要使用的时候每次读一行出来即可。

下面的例子演示了 `with-open` 和 `line-seq` 的用法。 它读出一个文件里面所有的行， 并且打印出包含某个关键字的那些行。

```
(use '1)

(defn print-if-contains [line word]
  (when (.contains line word) (println line)))

(let [file "story.txt"
      word "fur"]

  ; with-open will close the reader after
  ; evaluating all the expressions in its body.
  (with-open [rdr (reader file)]
    (doseq [line (line-seq rdr)] (print-if-contains line word))))
```

`slurp` 函数把一个文件里面的所有的内容读进一个字符串里面并且返回。
`spit` 把一个字符串写进一个文件里面然后关闭这个文件。

这篇文章只是大概过了一下clojure的io里面提供了哪些函数来进行I/O操作。大家可以看下clojure源文件： `clojure/java/io.clj` 以了解其它一些函数。

解构

解构可以用在一个函数或者宏的参数里面来把一个集合里面的一个或者几个元素抽取到一些本地binding里面去。它可以用在由 `let` special form 或者 `binding` 宏所创建的binding里面。

比如，如果我们有一个vector或者一个list， 我们想要获取这个集合里面的第一个元素和第三个元素的和。那么可以用下面两种办法， 第二种解构的方法看起来要简单一点。

```
(defn approach1 [numbers]
  (let [n1 (first numbers)
   n3 (nth numbers 2)]
    (+ n1 n3)))

; Note the underscore used to represent the
; second item in the collection which isn't used.
(defn approach2 [[n1 _ n3]] (+ n1 n3))

(approach1 [4 5 6 7]) ; -> 10
(approach2 [4 5 6 7]) ; -> 10
```

&符合可以在解构里面用来获取集合里面剩下的元素。比如：

```
(defn name-summary [[name1 name2 & others]]
  (println (str name1 ", " name2) "and" (count others) "others"))

(name-summary ["Moe" "Larry" "Curly" "Shemp"]) ; -> Moe, Larry and
```

`:as` 关键字可以用来获取对于整个被解构的集合的访问。如果我们想要一个函数接受一个集合作为参数，然后要计算它的第一个元素与第三个元素的和占总和的比例，看下面的代码：

```
(defn first-and-third-percentage [[n1 _ n3 :as coll]]
  (/ (+ n1 n3) (apply + coll)))

(first-and-third-percentage [4 5 6 7]) ; ratio reduced from 10/22 -
```

解构也可以用来从map里面获取元素。假设我们有一个map这个map的key是月份，value对应的是这个月的销售额。那么我们可以写一个函数来计算夏季的总销售额占全年销售额的比例：

```
(defn summer-sales-percentage
  ; The keywords below indicate the keys whose values
  ; should be extracted by destructuring.
  ; The non-keywords are the local bindings
  ; into which the values are placed.
  [{june :june july :july august :august :as all}]
  (let [summer-sales (+ june july august)
        all-sales (apply + (vals all))]
    (/ summer-sales all-sales)))

(def sales {
  :january   100 :february 200 :march      0 :april    300
  :may       200 :june     100 :july     400 :august   500
  :september 200 :october  300 :november 400 :december 600})

(summer-sales-percentage sales) ; ratio reduced from 1000/3300 -> 1
```

我们一般使用和map里面key的名字一样的本地变量来对map进行解构，比如上面例子里面我们使用的 `{june :june july :july august :august :as all}` . 这个可以使用 `:keys` 来简化。比如, `{:keys [june july august] :as all}` .

命名空间

Java用class来组织方法， 用包来组织class。Clojure用名字空间来组织事物。“事物”包括Vars, Refs, Atoms, Agents, 函数, 宏 以及名字空间本身。

符号(Symbols)是用来给函数、宏以及binding来分配名字的。符号被划分到名字空间里面去了。 任何时候总有一个默认的名字空间，初始化的时候这个默认的名字空间是“user”，这个默认的名字空间的值被保存在特殊符号 `*ns*` .里面。默认的名字空间可以通过两种方法来改变。 `in-ns` 函数只是改变它而已. 而 `ns` 宏则做得更多。其中一件就是它会使得 `clojure.core` 名字空间里面的符号在新的名字空间里面都可见 (使用 `refer` 命令). `ns` 宏的其它一些特性我们会在后面介绍。

"user" 这个名字空间提供对于 `clojure.core` 这个名字空间里面所有符号的访问。同样道理对于那些通过 `ns` 宏来改变成默认名字空间的名字空间里面也是可以看到 clojure.core里面的所有的函数的。

如果要访问哪些不在默认名字空间里面的符号、函数， 那么你必须要指定全限定的完整名字。比如 clojure.string 包里面定义了一个 `join` 函数。它把多个字符串用一个分隔符隔开然后连起来，返回这个连起来的字符串。它的全限定名是 `clojure.string/join` .

`require` 函数可以加载 Clojure 库。它接受一个或者多一个名字空间的名字(注意前面的单引号)

```
(require 'clojure.string)
```

这个只会加载这个类库。这里面的名字还必须是一个全限定的报名， 包名之间用.分割。注意，clojure里面名字空间和方法名之间的分隔符是/而不是java里面使用的.。比如：

```
(clojure.string/join "$" [1 2 3]) ; -> "1$2$3"
```

`alias` 函数给一个名字空间指定一个别名以减少我们打字工作。当然这个别名的定义只在当前的名字空间里面有效。比如：

```
(alias 'su 'clojure.string)
(su/join "$" [1 2 3]) ; -> "1$2$3"
```

`refer` 函数使得指定的名字空间里面的函数在当前名字空间里面可以访问(不用使用全限定名字)。一个特例就是如果当前名字空间有那个名字空间一样的名字， 那么你访问的时候还是要制定名字空间的。看例子：

```
(refer 'clojure.string)
```

现在，上面的代码可以写成。

```
(join "$" [1 2 3]) ; -> "1$2$3"
```

我们通常把 `require` 和 `refer` 结合使用, 所以clojure提供了一个 `use` ，它相当于require和refer的简洁形式。

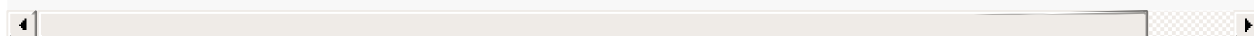
```
(use 'clojure.string)
```

`ns` 宏, 可以改变当前的默认名字空间。我们通常在一个源代码的最上面指定这个。它支持这些指令： `:require` ， `:use` 和 `:import` (用来加载 Java 类的) 这些其实是它们对应的函数的另外一种方式。我们鼓励使用这些指令而不是那些函数。 在下面的例子里面 注意 `:as` 给名字空间创建了一个别名。同时注意使用 `:only` 指令来加载Clojure库的一部分。

```
(ns com.ociweb.demo
  (:require 1)
  ; assumes this dependency: [org.clojure/math.numeric-tower "0.0.1
  (:use 1)
  (:import (java.text NumberFormat) (javax.swing JFrame JLabel)))

(println (su/join "$" [1 2 3])) ; -> 1$2$3
(println (gcd 27 72)) ; -> 9
(println (sqrt 5)) ; -> 2.23606797749979
(println (.format (NumberFormat/getInstance) Math/PI)) ; -> 3.142

; See the screenshot that follows this code.
(<a name="doto">doto</a> (JFrame. "Hello")
  (.add (JLabel. "Hello, World!"))
  (.pack)
  (.setDefaultCloseOperation JFrame/EXIT_ON_CLOSE)
  (.setVisible true))
```





`create-ns` 函数可以创建一个新的名字空间。但是不会把它变成默认的名字空间。 `def` 在当前名字定义一个符号，你同时还可以给它一个初始值。 `intern` 函数在一个指定名字空间里面定义一个符号(如果这个符号不存在的话)， 同时还可以给它指定一个默认值。注意在 `intern` 里面符号的名字要括起来，但是在 `def` 里面不需要。这是因为 `def` 是一个 special form, special form 不会 evaluate它的参数, 而 `intern` 是一个函数， 它会evaluate它的参数。看例子：

```
(def foo 1)
(create-ns 'com.ociweb.demo)
(intern 'com.ociweb.demo 'foo 2)
(println (+ foo com.ociweb.demo/foo)) ; -> 3
```

`ns-interns` 函数返回一个指定的名字空间的所有的符号的map(这个名字空间一定要在当前名字空间里面加载了), 这个map的key是符号的名字， value是符号所对应的 `Var` 对象， 这个对象表示的可能是函数，宏或者binding。 比如：

```
(ns-interns 'clojure.math.numeric-tower)
```

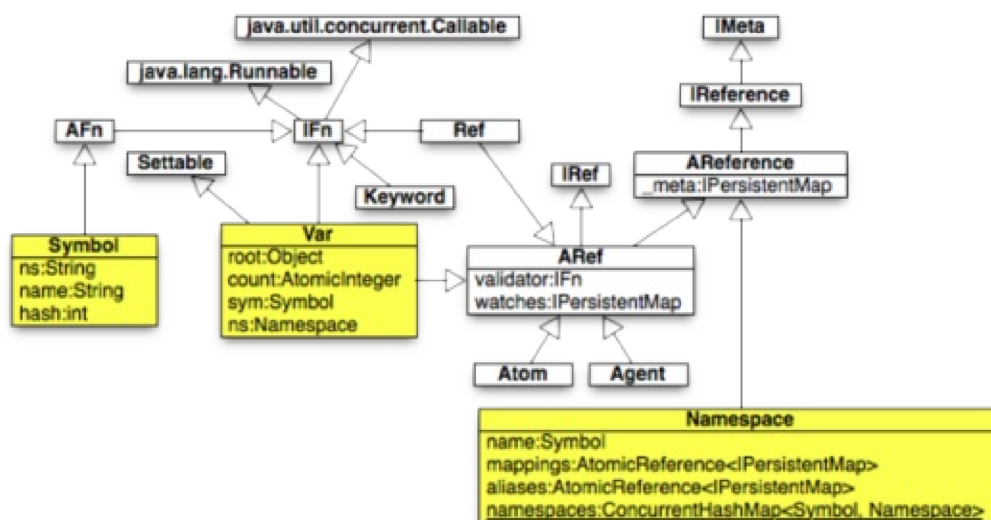
`all-ns` 函数返回一个包含当前所有的已经加载了的名字空间的集合。下面这些名字空间是默认加载的: `clojure.core` , `clojure.main` , `clojure.set` , `clojure.xml` , `clojure.zip` 以及 `user` . 而如果是在用REPL的话， 那么下面这些名字空间也会被加载： `clojure.repl` 和 `clojure.java.javadoc` .

`namespace` 函数返回一个给定符号或者关键字的名字空间。

其它一些在这里没有讨论的名字空间相关的函数还包括 `ns-aliases` , `ns-imports` , `ns-map` , `ns-name` , `ns-publics` , `ns-refers` , `ns-unalias` , `ns-unmap` 和 `remove-ns` .

Some Fine Print

`Symbol` 对象有一个 `String` 类型的名字以及一个 `String` 类型的名字空间名字(叫做 `ns`), 但是没有值。它使用一个字符串的名字空间而不是一个名字空间对象使得它可以指向一个还不存在的名字空间。 `Var` 对象有一个执行 `Symbol` 对象的引用 (叫做 `sym`), 一个指向 `Namespace` 对象的引用 (叫做 `ns`) 以及一个 `Object` 类型的对象(也就是它的root value, 叫做 `root`). `Namespace` 对象bjects有一个指向 `Map` 的引用， 这个map维护 `Symbol` 对象和 `Var` 对象的对应关系 (叫做 `mappings`)。同时它还有一个map来维护 `Symbol` 别名和 `Namespace` 对象之间的关系 (叫做 `namespaces`). 下面这个类图显示了Java里面的类和接口在Clojure里面的实现。在Clojure里面 "interning" 这个单词一般指的是添加一个 `Symbol` 到 `Var` 的对应关系到一个 `Namespace` 里面去。



元数据

Clojure里面的元数据是附加到一个符号或者集合的一些数据，它们和符号或者集合的逻辑数据没有直接的关系。两个逻辑上一样的方法可以有不同的元数据。 下面是一个有关扑克牌的例子

```
(defstruct card-struct :rank :suit)

(def card1 (struct card-struct :king :club))
(def card2 (struct card-struct :king :club))

(println (== card1 card2)) ; same identity? -> false
(println (= card1 card2)) ; same value? -> true

(def card2 #^{:bent true} card2) ; adds metadata at read-time
(def card2 (with-meta card2 {:bent true})) ; adds metadata at run-t
(println (meta card1)) ; -> nil
(println (meta card2)) ; -> {:bent true}
(println (= card1 card2)) ; still same value despite metadata diff
```

一些元数据是Clojure内部定义的。比如 `:private` 它表示一个Var是否能被包外的函数访问。 `:doc` 是一个 Var 的文档字符串。 `:test` 元数据是一个Boolean值表示这个函数是否是一个测试函数。

`:tag` 是一个字符串类型的类名或者一个 `Class` 对象，表示一个Var在Java里面对应的类型，或者一个函数的返回值。这些被称为“类型提示”。提供这些可以提高代码性能。如果你想查看你的clojure代码里面哪里使用反射来决定类型信息 -- 也就是说这里可能会有性能的问题， 那么你可以设置全局变量 `*warn-on-reflection*` 为 `true` 。

一些元数据会由Clojure的编译器自动地绑定到Var对象。 `:file` 是定义这个 Var 的文件的名字。 `:line` 是定义这个Var的行数。 `:name` 是一个Var的名字的 `Symbol` 对象。 `:ns` 是一个 `Namespace` 对象描述这个Var所在的名字空间。 `:macro` 是一个标识符标识这个符号是不是一个宏。 `:arglist` 是一个装有一堆vector的一个list， 表示一个函数所接受的所有的参数列表(前面在介绍函数的时候说过一个函数可以接受多个参数列表)。

函数以及宏，都是有一个 `Var` 对象来表示的, 它们都有关联的元数据。比如输入这个在REPL里面： `(meta (var reverse))` 或者 `^#'reverse` 。输出结果应该下面这些类似(为了好看我加了换行缩进)

```
{
  :ns #<Namespace clojure.core>,
  :name reverse,
  :file "core.clj",
  :line 630,
  :arglists ([coll]),
  :doc "Returns a seq of the items in coll in reverse order. Not la
}
```

clojure.repl包里面的 `source` 函数，利用元数据来获取一个指定函数的源代码，比如：

```
(source reverse)
```

上面代码的输出应该是：

```
(defn reverse
  "Returns a seq of the items in coll in reverse order. Not lazy."
  [coll]
    (reduce conj nil coll))
```

宏

宏是用来给语言添加新的结构，新的元素的。它们是一些在读入期（而不是编译期）就会实际代码替换的一个机制。

对于函数来说，它们的所有的参数都会被evaluate的, 而宏则会自动判断哪些参数需要evaluate。 这对于实现像 `(if _condition_ _then-expr_ _else-expr_)` 这样的结构是非常重要的。 如果 condition 是 `true` , 那么只有 "then" 表达式需要被evaluated. 如果条件是 `false` , 那么只有 "else" 表达式应该被 evaluated. 这意味着 `if` 不能被实现成一个函数 (它其实也不是宏， 而是一个special form)。其它一些因为这个原因而必须要实现成宏的包括 `and` 和 `or` 因为它们需要实现 "short-circuit"属性。

要想知道一个东西到底是函数还是宏， 可以在REPL里面输入 `(doc _name_)` 或者查看它的元数据。如果是一个宏的话，那么它的元数据里面包含一个 `:macro` key， 并且它的值为 `true` 。 比如，我们要看看 `and` , 是不是宏， 在REPL里面输入下面的命令:

```
((meta (var and)) :macro) ; long way -> true
(^#'and :macro) ; short way -> true
```

让我们通过一些例子来看看如何编写并且使用宏。假设我们代码里面很多地方要对一个数字进行判断，通过判断它是接近0， 是正的， 是负的来执行不同的逻辑；我们又不想这种判断的代码到处重复，那么这种情况下我们就可以使用宏了。我们使用 `defmacro` 宏来定义一个宏。

```
(defmacro around-zero [number negative-expr zero-expr positive-expr
  `(let [number# ~number] ; so number is only evaluated once
    (cond
      (< (Math/abs number#) 1e-15) ~zero-expr
      (pos? number#) ~positive-expr
      true ~negative-expr)))
```

Clojure的reader会把所有调用around-aero的地方全部换成defmacro这个方法体里面的具体代码。我们在这里使用let是为了性能，因为这个传进来的number是一个表达式而不是一个简单的值， 而且被cond语句里面使用了两次。自动产生的变量number#是为了产生一个不会和用户指定的其它binding冲突的一个名字。这使得我们可以创建 hygienic macros .

宏定义开始的时候的那个反引号 (也称为语法引号) 防止宏体内的任何一个表达式被evaluate -- 除非你显示地转义了。这意味着宏体里面的代码会原封不动地替换到使用这个宏的所有的地方 -- 除了以波浪号开始的那些表达式。 (`number` ,

`zero-expr` , `positive-expr` 和 `negative-expr`). 当一个名字前面被加了一个波浪号，并且还在反引号里面，它的值会被替换的。如果这个名字代表的是一个序列，那么我们可以用 `~@` 这个语法来替换序列里面的某个具体元素。

下面是两个使用这个宏的例子：(输出都应该是 " `+` ").

```
(around-zero 0.1 (println "-") (println "0") (println "+"))
(println (around-zero 0.1 "-" "0" "+")) ; same thing
```

如果对于每种条件执行多于一个表达式， 那么用do把他们包起来。看下面例子：

```
(around-zero 0.1
  (do (log "really cold!") (println "-"))
  (println "0")
  (println "+"))
```

为了验证这个宏是否被正确展开， 在REPL里面输入这个：

```
(macroexpand-1
  '(around-zero 0.1 (println "-") (println "0") (println "+")))
```

它会输出下面这个(为了容易看懂， 我加了缩进)

```
(clojure.core/let [number__3382__auto__ 0.1]
  (clojure.core/cond
    (clojure.core/< (Math/abs number__3382__auto__) 1.0E-15) (print
    (clojure.core/pos? number__3382__auto__) (println "+")
    true (println "-")))
```

下面是一个使用这个宏来返回一个描述输入数字的属性的字符串的函数。

```
(defn number-category [number]
  (around-zero number "negative" "zero" "positive"))
```

下面是一些示例用法：

```
(println (number-category -0.1)) ; -> negative
(println (number-category 0)) ; -> zero
(println (number-category 0.1)) ; -> positive
```

因为宏不会 evaluate 它们的参数, 所以你可以在宏体里面写一个对函数的参数调用. 函数定义不能这么做，相反只能用匿名函数把它们包起来。

下面是一个接受两个参数的宏。第一个是一个接受一个参数的函数, 这个参数是一个弧度， 如果它是一个三角函数sin， cos。第二个参数是一个弧度。如果这个被写成一个函数而不是一个 宏的话， 那么我们需要传递一个 `#(Math/sin %)` 而不是简单的 `Math/sin` 作为参数。注意 那些后面的#符号， 它会产生一个唯一的、不冲突的本地binding。 `#` 和 `~` 都必须在反引号引着的列表里面才能使用。

```
(defmacro trig-y-category [fn degrees]
  `(let [radians# (Math/toRadians ~degrees)
         result# (~fn radians#)]
     (number-category result#)))
```

让我们试一下。下面代码的期望输出应该是 "zero", "positive", "zero" 和 "negative".

```
(doseq [angle (range 0 360 90)] ; 0, 90, 180 and 270
  (println (trig-y-category Math/sin angle)))
```

宏的名字不能作为参数传递给函数。比如一个宏的名字比如 `and` 不能作为参数传递给 `reduce` 函数。一个绕过的方法是定义一个匿名函数把这个宏包起来。比如 `(fn [x y] (and x y))` 或者 `#(and %1 %2)` . 宏会在这个读入期在这个匿名函数体内解开。当这个函数被传递给函数比如 `reduce` , 传递的是函数而不是宏。

宏的调用是在读入期处理的。

并发

Wikipedia上面对于并发有个很精准的定义：

"Concurrency is a property of systems in which several computations are executing and overlapping in time, and potentially interacting with each other. The overlapping computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors."

并发编程的主要挑战就在于管理对于共享的、可修改的状态的修改。

通过锁来对并发进行管理是非常难的。我们需要决定哪些对象需要加锁以及什么时候加锁。这还不算完， 每次你修改代码或者添加新的代码的时候你都要重新审视下你的这些决定。如果一个开发人员忘记了去锁一个应该加锁的对象，或者锁的时机不对，一些非常糟糕的事情就会发生了。这些糟糕的事情包括 死锁 和 竞争条件 ；另一个方面如果你锁了一个不需要锁的对象，那么你的系统的性能则会下降。

为了更好地进行并发编程是很多开发人员选择Clojure的原因。Clojure的所有的数据都是只读的，除非你显示的用Var, Ref ,Atom和Agent来标明它们是可以修改的。这些提供了安全的方法去管理共享状态，我们会在下一节："引用类型"里面更加详细地介绍。

用一个新线程来运行一个Clojure函数是非常简单的，不管它是内置的，还是自定义的， 不管它是有名的还是匿名的。关于这个更详细的可以看上面有关和Java的互操作的讨论。

因为Clojure代码可以使用java里面的所有的类和接口， 所以它可以使用Java的并发能力。Java领域一个很棒的有关java并发编程的书： " Java Concurrency In Practice ". 这本书里面讲到了很多java里面如果做好并发编程的一些建议。但是要做到这些建议并不是一件很轻松的事情。在大多数情况下，使用java的引用类型比使用java里面并发要更简单。

除了引用类型， Clojure还提供了其它一些函数来使你的并发编程更简单。

`future` 宏把它的body里面的表达式在另外一个线程里面执行（这个线程来自于 `CachedThreadPool` ，Agents(后面会介绍)用的也是这个). 这个对于那种运行时间比较长， 而且一下子也不需要运行结果的程序来说比较有用。你可以通过dereferencing 从 `future` . 放回的对象来得到返回值。 如果计算已经结束了， 那么立马返回那个值；如果计算还没有结束，那么当前线程会block住，直到计算结束返回。因为这里使用了一个来自Agent线程池的线程, 所以我们要在一个适当的时机调用 `shutdown-agents` 关闭这些线程，然后程序才能退出。

为了演示 `future` 的用法， 我们加了一些println的方法调用，它能帮助我们观察方法执行的状态，注意输出的消息的顺序。

```
(println "creating future")
(def my-future (future (f-prime 2))) ; f-prime is called in another
(println "created future")
(println "result is" @my-future)
(shutdown-agents)
```

如果 `f-prime` 是一个比较耗时的方法的话, 那么输出应该是这样的：

```
creating future
created future
derivative entered
result is 9.0
```

`pmap` 函数把一个函数作用到一个集合里面的所有的元素， 和map不一样的是这个过程是完全并行的， 所以如果你要调用的这个函数是非常耗时间的话， 那么使用pmap将比使用

`clojure.parallel` 名字空间里头有好多方法可以帮助你并行化你的代码, 他们包括： `par` ，`pdistinct` ，`pfilter-dupes` ，`pfilter-nils` ，`pmax` ，`pmin` ，`preduce` ，`psort` ，`psummary` 和 `pvec` 。

引用类型

引用类型是一种可变引用指向不可变数据的一种机制。Clojure里面有4种引用类型:Vars,Refs,Atoms和Agents. 它们有一些共同的特征:

- 它们都可以指向任意类型的对象。
- 都可以利用函数 `deref` 以及宏 `@` 来读取它所指向的对象。
- 它们都支持验证函数，这些函数在它们所指向的值发生变化的时候自动调用。如果新值是合法的值，那么验证函数简单的返回true, 如果新值是不合法的，那么要么返回false， 要么抛出一个异常。如果只是简单地返回了 `false` , 那么一个 `IllegalStateException` 异常会被抛出，并且带着提示信息： "Invalid reference state" 。
- 如果是Agents的话，它们还支持watchers。如果被监听的引用的值发生了变化，那么Agent会得到通知， 详情见 "Agents" 一节。

下面的这个表格总结了一下四种引用类型的区别，以及分别要用什么方法去创建或者修改它们。这个表格里面提到的函数我们会在后面介绍。

	Var
目的	同步对于一个线程本地(thread-local)的变量的修改。
创建方法	`(def name initial-value)`
修改方法	`(def name new-value)` - 可以赋新的值 `(alter-var-root (var name) update-fn args)` - 自动设置新值 `(set! name new-value) - 在一个binding form 里满设置一个新的、线程本`

Vars

Vars 是一种可以有一个被所有线程共享的root binding并且每个线程线程还能有自己线程本地(thread-local)的值的一种引用类型。

下面的语法创建一个Var并且给它一个root binding:

```
(def <em>name</em> <em>value</em>)
```

你可以不给它一个值的。如果你没有给它一个值，那么我们说这个Var是"unbound". 同样的语法可以用来修改一个Var的root binding。

有两种方法可以创建一个已经存在的Var的线程本地binding(thread-local-binding):

```
(binding [<em>name</em> <em>expression</em>] <em>body</em>)
(set! <em>name</em> <em>expression</em>) ; inside a binding that bo
```

关于binding宏的用法我们前面已经介绍过了. 下面的例子演示把它和 `set!` 一起使用. 用set!来修改一个由binding bind的Var的线程本地的值。

```
(def v 1)

(defn change-it []
  (println "2) v =" v) ; -> 1

  (def v 2) ; changes root value
  (println "3) v =" v) ; -> 2

  (binding [v 3] ; binds a thread-local value
    (println "4) v =" v) ; -> 3

    (set! v 4) ; changes thread-local value
    (println "5) v =" v)) ; -> 4

  (println "6) v =" v)) ; thread-local value is gone now -> 2

(println "1) v =" v) ; -> 1

(let [thread (Thread. #(change-it))]
  (.start thread)
  (.join thread)) ; wait for thread to finish

(println "7) v =" v) ; -> 2
```

我们一般不鼓励使用 Vars， 因为线程之间对于同一个Var的修改没有做很好的协调，比如线程A在使用一个Var的root值，然后才发现，在它使用这个值的时候，已经有一个线程B在修改这个值了。

Refs

Refs是用来协调对于一个或者多个binding的并发修改的。这个协调机制是利用 Software Transactional Memory (STM)来实现的。 Refs指定在一个事务里面修改。

STM在某些方面跟数据库的事务很像。在一个STM事务里面做的修改只有在事务提交之后别的线程才能看到。这实现了ACID里面的A和I。Validation函数是的对Ref的修改与跟它相关的其它的值是一致的(consistent), 也就实现了C。

要想你的代码在一个事务里面执行， 那么要把你的代码包在宏 `dosync` 的体内。当在一个事务里面对值进行修改，被改的其实是一个私有的、线程内的、直到事务提交才会被别的线程看到的一快内存。

如果到事务结束的时候也没有异常抛出的话， 那么这个事务会顺利的提交， 在事务里面所作的改变也就可以被别的线程看到了。

如果在事务里面有一个异常抛出，包括validation函数抛出的异常，那么这个事务会被回滚，事务里面对值做的修改也就会撤销。

如果在一个事务里面，我们要对一个Ref进行修改，但是发现从我们的事务开始之后，已经有别的线程对这个Ref做了改动（冲突了）， 那么当前事务里面的改动会被撤销，然后从dosync的开头重试。那到底什么时候会检测到冲突， 什么时候会进行重试， 这个是没有保证的， 唯一保证的是clojure为检测到冲突，并且会进行重试。

要在事务里面执行的代码一定要是没有副作用的，这一点非常重要，因为前面提到的，事务可能会跟别的事务事务冲突，然后重试， 如果有副作用的话，那么出来的结果就不对了。不过要执行有副作用的代码也是可能的， 可以把这个方法调用包装给Agent, 然后这个方法会被hold住直到事务成功提交，然后执行一次。如果事务失败那么就不会执行。

`ref` 函数可以创建一个 Ref 对象。下面的例子代码创建一个Ref并且得到它的引用。

```
(def <em>name</em> (ref <em>value</em>))
```

`dosync` 宏用来包裹一个事务 -- 从它对应的左括号开始，到它对应的右括号结束。在事务里面我们用 `ref-set` 来改变一个Ref的值并且返回这个值。你不能在事务之外调用这个函数，否则会抛出 `IllegalStateException` 异常。 看例子：

```
(dosync
  ...
  (ref-set <em>name</em> <em>new-value</em>)
  ...)
```

如果你要赋的新值是基于旧的值的话，那么就需要三个步骤了：

1. deference 这个 Ref 来获得它的旧值
2. 计算新值
3. 设置新值

`alter` 和 `commute` 函数在一个操作里面完成这三个步骤。 `alter` 函数是用来操作那些必须以特定顺序进行的修改。而 `commute` 函数则是要来操作那些修改顺序不是很重要 -- 可以同时进行的修改。 跟 `ref-set` ,一样， 它们只能在一个事务里面调用。它们都接受一个 "update 函数" 做为参数， 以及一些额外的参数来计算新的值。这个函数会被传递这个Ref在线程内的当前的值以及一些额外的参数（如果有的话）。当我们要赋的新的值是基于旧的值计算出来的时候, 那么我们鼓励使用 `alter` 和 `commute` 而不是 `ref-set` .

比如，我们想给一个Ref: `counter` 加一， 我们可以用 `inc` 函数来实现：

```
(dosync
  ...
  (alter counter inc)
  ; or as
  (commute counter inc)
  ...)
```

如果 `alter` 试图修改的 Ref 在当前事务开始之后被别的事务改变了，那么当前事务会进行重试。而同样的情况下 `commute` 不会进行重试。它会以事务内的当前值进行计算。这会获得比较好的性能(因为不进行重试)。但是要记住的是 `commute` 函数只有在多个线程对Ref的修改顺序不重要的时候才能使用。

如果一个事务提交了， 那么对于 `commute` 函数还会有一些额外的事情发生。对于每一个 `commute` 调用, Ref 的值会被下面的调用结果重置:

```
(apply <em>update-function</em> <em>last-committed-value-of-ref</er
```

注意，这个update-function会被传递这个Ref最后被提交的值， 这个值可能是另外一个、在我们当前事务开始之后才开始的事务。

使用 `commute` 而不是 `alter` 是一种优化。只要对Ref进行更新的顺序不会影响到这个Ref的最终的值。

然后看一个使用了 Refs 和 Atoms (后面会介绍)的例子。这个例子涉及到银行账户以及账户之间的交易。首先我们定义一下数据模型。

```
(ns com.ociweb.bank)

; Assume the only account data that can change is its balance.
(defstruct account-struct :id  wner :balance-ref)

; We need to be able to add and delete accounts to and from a map.
; We want it to be sorted so we can easily
; find the highest account number
; for the purpose of assigning the next one.
(def account-map-ref (ref (sorted-map)))
```

下面的函数建立一个新的帐户，并且把它存入帐户的map, ? 然后返回它。

```
(defn open-account
  "creates a new account, stores it in the account map and returns
  [owner]
  (dosync ; required because a Ref is being changed
    (let [account-map @account-map-ref
     last-entry (last account-map)
     ; The id for the new account is one higher than the last one.
     id (if last-entry (inc (key last-entry)) 1)
     ; Create the new account with a zero starting balance.
     account (struct account-struct id owner (ref 0))]
  ; Add the new account to the map of accounts.
  (alter account-map-ref assoc id account)
  ; Return the account that was just created.
  account)))
```

下面的函数支持从一个账户里面存/取钱。

```
(defn deposit [account amount]
  "adds money to an account; can be a negative amount"
  (dosync ; required because a Ref is being changed
    (Thread/sleep 50) ; simulate a long-running operation
    (let [owner (account  wner)
          balance-ref (account :balance-ref)
          type (if (pos? amount) "deposit" "withdraw")
          direction (if (pos? amount) "to" "from")
          abs-amount (Math/abs amount)]
      (if (>= (+ @balance-ref amount) 0) ; sufficient balance?
        (do
          (alter balance-ref + amount)
          (println (str type "ing") abs-amount direction owner))
        (throw (IllegalArgumentException.
                 (str "insufficient balance for " owner
                      " to withdraw " abs-amount)))))))

(defn withdraw
  "removes money from an account"
  [account amount]
  ; A withdrawal is like a negative deposit.
  (deposit account (- amount)))
```

下面是函数支持把钱从一个账户转到另外一个账户。由 `dosync` 所开始的事务保证转账要么成功要么失败，而不会出现中间状态。

```
(defn transfer [from-account to-account amount]
  (dosync
    (println "transferring" amount
     "from" (from-account  wner)
     "to" (to-account  wner))
    (withdraw from-account amount)
    (deposit to-account amount)))
```

下面的函数支持查询账户的状态。由 `dosync` 所开始的事务保证事务之间的一致性。比如把不会报告一个转账了一半的金额。

```
(defn- report-1 ; a private function
  "prints information about a single account"
  [account]
  ; This assumes it is being called from within
  ; the transaction started in report.
  (let [balance-ref (account :balance-ref)]
    (println "balance for" (account  wner) "is" @balance-ref)))

(defn report
  "prints information about any number of accounts"
  [& accounts]
  (dosync (doseq [account accounts] (report-1 account))))
```

上面的代码没有去处理线程启动时候可能抛出的异常。相反，我们在当前线程给他们定义了一个异常处理器。

```
; Set a default uncaught exception handler
; to handle exceptions not caught in other threads.
(Thread/setDefaultUncaughtExceptionHandler
  (proxy [Thread$UncaughtExceptionHandler] []
    (uncaughtException [thread throwable]
      ; Just print the message in the exception.
      (println (.. throwable .getCause .getMessage)))))
```

现在我们可以调用上面的函数了。

```
(let [a1 (open-account "Mark")
   a2 (open-account "Tami")
   thread (Thread. #(transfer a1 a2 50))]
  (try
  (deposit a1 100)
  (deposit a2 200)

  ; There are sufficient funds in Mark's account at this point
  ; to transfer $50 to Tami's account.
  (.start thread) ; will sleep in deposit function twice!

  ; Unfortunately, due to the time it takes to complete the transfe
  ; (simulated with sleep calls), the next call will complete first
  (withdraw a1 75)

  ; Now there are insufficient funds in Mark's account
  ; to complete the transfer.

  (.join thread) ; wait for thread to finish
  (report a1 a2)
  (catch IllegalArgumentException e
   (println (.getMessage e) "in main thread"))))
```

上面代码的输出是这样的：

```
depositing 100 to Mark
depositing 200 to Tami
transferring 50 from Mark to Tami
withdrawing 75 from Mark
transferring 50 from Mark to Tami (a retry)
insufficient balance for Mark to withdraw 50
balance for Mark is 25
balance for Tami is 200
```

Validation函数

在继续介绍下一个引用类型之前，下面是一个validation函数的例子，他验证所有赋给Ref的值是数字。

```
; Note the use of the :validator directive when creating the Ref
; to assign a validation function which is integer? in this case.
(def my-ref (ref 0 :validator integer?))

(try
  (dosync
    (ref-set my-ref 1) ; works

    ; The next line doesn't work, so the transaction is rolled back
    ; and the previous change isn't committed.
    (ref-set my-ref "foo"))
  (catch IllegalStateException e
    ; do nothing
    ))

(println "my-ref =" @my-ref) ; due to validation failure -> 0
```

Atoms

Atoms 提供了一种比使用Refs&STM更简单的更新当个值的方法。它不受事务的影响

有三个函数可以修改一个Atom的值： `reset!` , `compare-and-set!` 和 `swap!` .

`reset!` 函数接受两个参数：要设值的Atom以及新值。它设置新的值，而不管你旧的值是什么。看例子：

```
(def my-atom (atom 1))
(reset! my-atom 2)
(println @my-atom) ; -> 2
```

`compare-and-set!` 函数接受三个参数：要被修改的Atom, 上次读取时候的值，新的值。 这个函数在设置新值之前会去读Atom现在的值。如果与上次读的时候的值相等， 那么设置新值并返回true, 否则不设置新值， 返回false。看例子：

```
(def my-atom (atom 1))

(defn update-atom []
  (let [curr-val @my-atom]
    (println "update-atom: curr-val =" curr-val) ; -> 1
    (Thread/sleep 50) ; give reset! time to run
    (println
      (compare-and-set! my-atom curr-val (inc curr-val))))) ; -> fa

(let [thread (Thread. #(update-atom))]
  (.start thread)
  (Thread/sleep 25) ; give thread time to call update-atom
  (reset! my-atom 3) ; happens after update-atom binds curr-val
  (.join thread)) ; wait for thread to finish

(println @my-atom) ; -> 3
```

为什么最后的结果是 3呢? `update-atom` 被放在一个单独的线程里面，在 `reset!` 函数调用之前执行。所以它获取了atom的初始值1(存到变量curr-val里面去了), 然后它sleep了以让 `reset!` 函数有执行是时间。在那之后，atom的值就变成3了。当 `update-atom` 函数调用 `compare-and-set!` 来给这个值加一的时候， 它发现atom的值已经不是它上次读取的那个值了(1), 所以更新失败， atom的值还是3。

`swap!` 函数接受一个要修改的 Atom, 一个计算Atom新值的函数以及一些额外的参数(如果需要的话)。这个计算Atom新的值的函数会以这个Atom以及一些额外的参数做为输入。swap！函数实际上是对compare-and-set!函数的一个封装，但是有一

个显著的不同。 它首先把Atom的当前值存入一个变量，然后调用计算新值的函数来计算新值， 然后再调用compare-and-set!函数来赋值。如果赋值成功的话，那就结束了。如果赋值不成功的话， 那么它会重复这个过程，一直到赋值成功为止。这就是它们的区别：所以上面的代码可以用swap!改写成这样：

```
(def my-atom (atom 1))

(defn update-atom [curr-val]
  (println "update-atom: curr-val =" curr-val)
  (Thread/sleep 50) ; give reset! time to run
  (inc curr-val))

(let [thread (Thread. #(swap! my-atom update-atom))]
  (.start thread)
  (Thread/sleep 25) ; give swap! time to call update-atom
  (reset! my-atom 3)
  (.join thread)) ; wait for thread to finish

(println @my-atom) ; -> 4
```

为什么输出变成4了呢？因为swap!会不停的去给curr-val加一直到成功为止。

Agents

Agents 是用把一些事情放到另外一个线程来做 -- 一般来说不需要事务控制的。它们对于修改一个单个对象的值(也就是Agent的值)来说很方便。这个值是通过在另外的一个thread上面运行一个“action”来修改的。一个action是一个函数，这个函数接受Agent的当前值以及一些其它参数。 Only one action at a time will be run on a given Agent在任意一个时间点一个Agent实例上面只能运行一个action.

`agent` 函数可以建立一个新的Agent. 比如:

```
(def my-agent (agent <em>initial-value</em>))
```

`send` 函数把一个 action 分配给一个 Agent，并且马上返回而不做任何等待。这个action会在另外一个线程(一般是由一个线程池提供的)上面单独运行。当这个action运行结束之后，返回值会被设置给这个Agent。 `send-off` 函数也类似只是线程来自另外一个线程吃。

`send` 使用一个 "固定大小的" 线程吃 (java.util.concurrent.Executors里面的newFixedThreadPool))，线程的个数是机器的处理器的个数加2。如果所有的线程都被占用，那么你如果要运行新的action，那你就要等了。 `send-off` 使用的是 "cached thread pool" (java.util.concurrent.Executors里面的?newCachedThreadPool))，这个线程池里面的线程的个数是按照需要来分配的。

如果 `send` 或者 `send-off` 函数是在一个事务里面被调用的。那么这个action直到线程提交的时候才会被发送给另外一个线程去执行。这在某种程度上来说和 `commute` 函数对 Ref 的作用是类似的。

在action里面，相关联的那个agent可以通过symbol： `*agent*` 得到。

`await` 以一个或者多个Agent作为参数，并且block住当前的线程，直到当前线程分派给这些Agent的action都执行完了。 `await-for` 函数是类似的, 但是它接受一个超时时间作为它的第一个参数，如果在超时之前事情都做完了，那么返回一个非nil的值，否则返回一个非nil的值，而且当前线程也就不再被block了。 `await` 和 `await-for` 函数不能在事务里面调用。

如果一个action执行的时候抛出一个异常了，那么你要dereference这个Agent的话也会抛出异常的。在action里面抛出的所有的异常可以通过 `agent-errors` 函数获取。 `clear-agent-errors` 函数可以清除一个指定Agent上面的所有异常。

`shutdown-agents` 函数等待所有发送给agents的action都执行完毕。然后它停止线程池里面所有的线程。在这之后你就不能发送新的action了。我们一定要调用 `shutdown-agents` 以让JVM 可以正常退出，因为Agent使用的这些线程不是守护线程，如果你不显式关闭的话，JVM是不会退出的。

Watchers

WARNING: 下面这个章节要做一些更新，因为在Clojure1.1里面 `add-watcher` 和 `remove-watcher` 这两个函数被去掉了。 两个不大一样的函数 `add-watch` 和 `remove-watch` 被添加进来了。

Agents 可以用作其它几种引用类型的监视器。当一个被监视的引用的值发生了改变之后，Clojure会通过给Agent发送一个action的形式通知它。通知的类型可以是 `send` 或者 `send-off` , 这个是在你把Agent注册为引用类型的监视器的时候指定的。那个action的参数是那个监视器 Agent 以及发生改变的引用对象。这个action的返回值则是Agent的新值。

就像我们前面已经说过的那样，函数式编程强调那种"纯函数" -- 不会改变什么全局变量的函数。但是Clojure也不绝对静止这样做， 但是Clojure使得我们要找出对全局状态进行了改变的函数非常的简单。一个方法就是寻找那些能对状态进行改变的宏和方法，比如 `alter` 。 这到了调用这些宏/函数的地方就找到了所有修改全局状态的地方了。另外一个方法就是用Agent来监视对于全局状态的更改。一个监视者可以通过dump出来stack trace来确定到底是谁对全局状态做了修改。

下面的例子给一个Var，一个Ref, 一个Atom注册了一个Agent监视者。Agent里面维护了它所监视的每个引用被修改的次数(一个map)。这个map的key就是引用对象，而值则是被修改的次数。

```
(def my-watcher (agent {}))

(defn my-watcher-action [current-value reference]
  (let [change-count-map current-value
        old-count (change-count-map reference)
        new-count (if old-count (inc old-count) 1)]
  ; Return an updated map of change counts
  ; that will become the new value of the Agent.
  (assoc change-count-map reference new-count)))

(def my-var "v1")
(def my-ref (ref "r1"))
(def my-atom (atom "a1"))

(add-watcher (var my-var) :send-off my-watcher my-watcher-action)
(add-watcher my-ref :send-off my-watcher my-watcher-action)
(add-watcher my-atom :send-off my-watcher my-watcher-action)

; Change the root binding of the Var in two ways.
(def my-var "v2")
(alter-var-root (var my-var) (fn [curr-val] "v3"))

; Change the Ref in two ways.
(dosync
  ; The next line only changes the in-transaction value
  ; so the watcher isn't notified.
  (ref-set my-ref "r2")
  ; When the transaction commits, the watcher is
  ; notified of one change this Ref ... the last one.
  (ref-set my-ref "r3"))
(dosync
  (alter my-ref (fn [_] "r4"))) ; And now one more.

; Change the Atom in two ways.
(reset! my-atom "a2")
(compare-and-set! my-atom @my-atom "a3")

; Wait for all the actions sent to the watcher Agent to complete.
(await my-watcher)

; Output the number of changes to
; each reference object that was watched.
(let [change-count-map @my-watcher]
  (println "my-var changes =" (change-count-map (var my-var))) ; ->
  (println "my-ref changes =" (change-count-map my-ref)) ; -> 2
  (println "my-atom changes =" (change-count-map my-atom))) ; -> 2

(shutdown-agents)
```

编译

当clojure的源代码文件被当作脚本文件执行的时候，它们是在运行时被编译成java的bytecode的。同时我们也可以提前编译(AOT ahead-of-time)它们成java bytecode。这会缩短clojure程序的启动时间，并且产生的.class文件还可以给java程序使用。我们推荐按照下面的步骤来做：

1. 为你要编译的文件选择一个名字空间，比如： `com.ociweb.talk` 。
2. 在父目录里面创建两个目录:" `src` "和" `classes` "。
3. 使你的其中一个文件的文件名和包名的最后一段相同，比如： `talk.clj` 。
4. 把你的源文件放在" `src` "目录下面，并且创建和名字空间一样的目录层级，比如： `src/com/ociweb/talk.clj` 。
5. 在你的源代码的最上面给你的文件指定名字空间，并且包含:gen-class标记： `(ns com.ociweb.talk (:gen-class))`
6. 在你的主源文件里面，使用 `load` 函数来加载同一个目录下面的其它源文件，比如，如果 `more.clj` 在目录 `src/com/ociweb` 的子目录" `talk` "下面那么用这个语句来加载 `(load "talk/more")` .
7. 在其它的源文件里面, 使用 `in-ns` 函数来设置他们的名字空间. 比如, 在more.clj文件上面指定名字空间: `(in-ns 'com.ociweb.talk)。`
8. 把" `src` "和" `classes` "目录添加到REPL的classpath里面去。如果你使用了一个脚本来运行REPL, 那么修改那个脚本。
9. 启动一个REPL。
10. 使用 `compile` 函数来编译一个给定名字空间的clojure文件: `(compile '_namespace_)。比如` : `(compile 'com.ociweb.talk)` .

这些步骤会为每个函数创建一个单独的.class文件。他们会被写到" `classes` "文件夹下对应的子文件夹下面去。

如果这个被编译的名字空间有一个叫做- `main` 的函数，那么你可以把它当作java的主类的运行。命令行参数会被当作参数传递给这个函数。比如，如果 `talk.clj` 包含一个叫 `-main` 的函数，你可以用下面的命令来运行:

```
java -classpath <em>path</em>/classes:<em>path</em>/clojure.jar cor
```

在Java里面调用 Clojure

提前编译的Clojure函数如果是静态的函数的话，那么它们可以被java程序调用。可以通过把函数的元数据项： `:static` 设置为 `true` 来达到这个目的。语法是这样的：

```
(ns <em>namespace</em>
  (:gen-class
   :methods [#^{:static true} [<em>function-name</em> [<em>param-ty
```

让我们看一个例子：下面是一个名字叫做Demo.clj的文件，它的路径是 `src/com/ociweb/clj`。

```
(ns com.ociweb.clj.Demo
  (:gen-class
   :methods [#^{:static true} [getMessage [String] String]]))

# Note the hyphen at the beginning of the function name!
(defn -getMessage [name]
  (str "Hello, " name "!"))
```

下面是一个叫做 `Main.java` 的java文件，它和 `src` 以及 `classes` 在同一个目录。

```
import com.ociweb.clj.Demo; // class created by compiling Clojure s

public class Main {

    public static void main(String[] args) {
        String message = Demo.getMessage("Mark");
        System.out.println(message);
    }
}
```

下面是编译并且运行它的步骤：

1. cd到包含 `src` 和 `classes` 的目录。
2. 通过" `clj` "命令来打开一个REPL。
3. 运行" `(compile 'com.ociweb.clj.Demo)` ".
4. 退出 REPL (ctrl-d 或者 ctrl-c).
5. 运行" `javap -classpath classes com.ociweb.clj.Demo` "来查看生成的class文件里面的方法。
6. 运行" `javac -cp classes Main.java` ".

7. 运行" `java -cp .:classes:_path_/clojure.jar Main.java` ". 注意Windows下面的路径分隔符是分号而不是冒号。
8. 输出应该是" `Hello, Mark!` ".

Clojure还有一些更加高级的编译特性。更多细节可以参考宏 `[gen-class](http://clojure.github.com/clojure/clojure.core-api.html` 的文档以及 http://clojure.org/compilation/ 。

自动化测试

Clojure里面主要的主要自动化测试框架是clojure core里面自带的。下面的代码演示了它的一些主要特性：

```
(use 'clojure.test)

; Tests can be written in separate functions.
(deftest add-test
  ; The "is" macro takes a predicate, arguments to it,
  ; and an optional message.
  (is (= 4 (+ 2 2)))
  (is (= 2 (+ 2 0)) "adding zero doesn't change value"))

(deftest reverse-test
  (is (= [3 2 1] (reverse [1 2 3]))))

; Tests can verify that a specific exception is thrown.
(deftest division-test
  (is (thrown? ArithmeticException (/ 3.0 0))))

; The with-test macro can be used to add tests
; to the functions they test as metadata.
(with-test
  (defn my-add [n1 n2] (+ n1 n2))
  (is (= 4 (my-add 2 2)))
  (is (= 2 (my-add 2 0)) "adding zero doesn't change value"))

; The "are" macro takes a predicate template and
; multiple sets of arguments to it, but no message.
; Each set of arguments are substituted one at a time
; into the predicate template and evaluated.
(deftest multiplication
  (are [n1 n2 result]
    (= (* n1 n2) result) ; a template
    1 1 1,
    1 2 2,
    2 3 6))

; Run all the tests in the current namespace.
; This includes tests that were added as function metadata using wi
; Other namespaces can be specified as quoted arguments.
(run-tests)
```

为了限制运行一个test的时候抛出来的异常的深度，bind一个数字到special symbol: `*stack-trace-depth*` 。

当你要把Clojure代码编译成bytecode以部署到生成环境的时候， 你可以给 `*load-tests*` symbol bind一个 `false` 值，以避免把测试代码编译进去。

虽然和自动化测试不是同一个层面的东西，还是值得一提的是Clojure提供一个宏： `assert` 。它测试一个表达式， 如果这个表达式的值为false的话，他们它会抛出异常。这可以警告我们这种情况从来都不应该发生。看例子:

```
(assert (>= dow 7000))
```

自动化测试的另外一个重要的特性是fixtures。fixture其实就是JUnit里面的setup和tearDown方法。fixture分为两种，一种是在每个测试方法的开始，结束的时候 执行。一种是在整个测试（好几个测试方法）的开始和结束的时候执行。

照下面的样子编写fixture:

```
(defn fixture-name [test-function]
  ; Perform setup here.
  (test-function)
  ; Perform teardown here.
)
```

这个fixture函数会在执行每个测试方法的时候执行一次。这里这个 `test-function` 及时要被执行的测试方法。

用下面的方法去注册这些fixtures去包裹每一个测试方法:

```
(use-fixtures :each fixture-1 fixture-2 ...)
```

执行的顺序是:

1. fixture-1 setup
2. fixture-2 setup
3. ONE test function
4. fixture-2 teardown
5. fixture-1 teardown

用下面的方法去注册这些fixtures去包裹整个一次测试:

```
(use-fixtures  nce fixture-1 fixture-2 ...)
```

执行顺序是这样的:

1. fixture-1 setup
2. fixture-2 setup
3. ALL test functions
4. fixture-2 teardown

5. fixture-1 teardown

Clojure本身的测试在 `test` 子目录里面. 要想运行他们的话, cd到包含 `src` 和 `test` 的目录下面去，然后执行："`ant test`"。

编辑器和IDE

Clojure plugins for many editors and IDEs are available. For emacs there is clojure-mode and swank-clojure, both at https://github.com/technomancy/swank-clojure . swank-clojure uses the Superior Lisp Interaction Mode for Emacs (Slime) described at http://common-lisp.net/project/slime/ . For Vim there is VimClojure http://kotka.de/projects/clojure/vimclojure.html . For NetBeans there is enclojure at http://enclojure.org/ . For IDEA there a "La Clojure" at http://plugins.intellij.net/plugin/?id=4050 . For Eclipse there is Counter Clockwise at http://dev.clojure.org/display/doc/Getting+Started+with+Eclipse+and+Counterclockwise .

桌面应用

Clojure 可以创建基于Swing的GUI程序。下面是一个简单的例子， 用户可以输入他们的名字，然后点击“Greet：按钮，然后它会弹出一个对话框显示一个欢迎信息。可以关注一下这里我们使用了 `proxy` 宏来创建一个集成某个指定类(`JFrame`)并且实现了一些java接口(这里只有 `ActionListener` 一个接口)的对象。.





```
(ns com.ociweb.swing
  (:import
    (java.awt BorderLayout)
    (java.awt.event ActionListener)
    (javax.swing JButton JFrame JLabel JOptionPane JPanel JTextFiel

(defn message
  "gets the message to display based on the current text in text-fi
  1
  (str "Hello, " (.getText text-field) "!"))

; Set the initial text in name-field to "World"
; and its visible width to 10.
(let [name-field (JTextField. "World" 10)
      greet-button (JButton. "Greet")
      panel (JPanel.)
      frame (proxy [JFrame ActionListener]
        [] ; superclass constructor arguments
        (actionPerformed [e] ; nil below is the parent component
          (JOptionPane/showMessageDialog nil (message name-field))
  (doto panel
    (.add (JLabel. "Name:"))
    (.add name-field))
  (doto frame
    (.add panel BorderLayout/CENTER)
    (.add greet-button BorderLayout/SOUTH)
    (.pack)
    (.setDefaultCloseOperation JFrame/EXIT_ON_CLOSE)
    (.setVisible true))
  ; Register frame to listen for greet-button presses.
  (.addActionListener greet-button frame))
```

Web应用

有很多Clojure类库可以帮助我们创建web应用。现在比较流行使用Chris Granger写的 Noir 。另外一个简单的，基于MVC的框架， 使用Christophe Grand写的? Enlive 来做页面的template, 是Sean Corfield写的 Framework One 。另一个流行的选择是James Reeves写的Compojure，你可以在这里下载：http://github.com/weavejester/compojure/tree/master 。所有这些框架都是基于Mark McGranahan写的 Ring (James Reeves同学现在在维护). 我们以Compojure为例子来稍微介绍一下web应用开发。最新的版本可以通过git来获取:

```
git clone git://github.com/weavejester/compojure.git
```

这个命令会在当前目录创建一个叫做 `compojure` 的目录. 另外你还需要从 http://cloud.github.com/downloads/weavejester/compojure/deps.zip 下载所有依赖的JAR包，把 `deps.zip` 下载之后解压在 `compojure` 目录里面的 `deps` 子目录里面：

要获取 `compojure.jar` , 在compojure里面运行 `ant` 命令。

要获取 Compojure的更新, 切换到 `compojure` 目录下面执行下面的命令:

```
git pull
ant clean deps jar
```

所有的 `deps` 目录里面的jar包都必须包含在classpath里面。一个方法是修改我们的 `clj` 脚本，然后用这个脚本来运行web应用. 把 " `-cp $CP` " 添加到 `java` 命令后面去 执行 `clojure.main添加下面这些行到脚本里面去，以把那些jar包包含在` `CP` 里面。

```
# Set CP to a path list that contains clojure.jar
# and possibly some Clojure contrib JAR files.
COMPOJURE_DIR=<em>path-to-compojure-dir</em>
COMPOJURE_JAR=$COMPOJURE_DIR/compojure.jar
CP=$CP:$COMPOJURE_JAR
for file in $COMPOJURE_DIR/deps/*.jar
do
  CP=$CP:$file
done
```

下面是他一个简单的 Compojure web应用：

Name: World
Reset Greet
Hello, World!

```
(ns com.ociweb.hello
  (:use compojure))

(def host "localhost")
(def port 8080)
(def in-path "/hello")
(def out-path "/hello-out")

(defn html-doc
  "generates well-formed HTML for a given title and body content"
  [title & body]
  (html
    (doctype :html4)
    [:html
      [:head [:title title]]
      [:body body]]))

; Creates HTML for input form.
(def hello-in
  (html-doc "Hello In"
    (form-to [:post out-path]
      "Name: "
      (text-field {:size 10} :name "World")
      [:br]
      (reset-button "Reset")
      (submit-button "Greet"))))

; Creates HTML for result message.
(defn hello-out [name]
  (html-doc "Hello Out"
    [:h1 "Hello, " name "!"]))

(defroutes hello-service
  ; The following three lines map HTTP methods
  ; and URL patterns to response HTML.
  (GET in-path hello-in)
  (POST out-path (hello-out (params :name)))
  (ANY "*" (page-not-found))) ; displays ./public/404.html by defau

(println (str "browse http://" host ":" port in-path))
; -> browse http://localhost:8080/hello
(run-server {:port port} "/*" (servl
```

数据库

Clojure Contrib里面的jdbc库简化了clojure对于关系型数据库的访问. 它通过commit和rollback来支持事务, 支持prepared statements, 支持创建/删除表, 插入/更新/删除行, 以及查询。下面的例子链接到一个Postgres 数据库并且执行了一个查询。代码的注释里面还提到了怎么使用jdbc来连接mysql。

```
(use 'clojure.java.jdbc)

(let [db-host "localhost"
      db-port 5432 ; 3306
      db-name "HR"]

  ; The classname below must be in the classpath.
  (def db {:classname "org.postgresql.Driver" ; com.mysql.jdbc.Driv
           :subprotocol "postgresql" ; "mysql"
           :subname (str "//" db-host ":" db-port "/" db-name)
           ; Any additional map entries are passed to the driver
           ; as driver-specific properties.
           :user "mvolkmann"
           :password "cljfan"})

  (with-connection db ; closes connection when finished
    (with-query-results rs ["select * from Employee"] ; closes resu
      ; rs will be a non-lazy sequence of maps,
      ; one for each record in the result set.
      ; The keys in each map are the column names retrieved and
      ; their values are the column values for that result set row
      (doseq [row rs] (println (row :lastname))))))
```

`clj-record` 提供了一个类似 Ruby on Rails的ActiveRecord的数据库访问包. 更多关于它的信息看这里： http://github.com/duelinmarkers/clj-record/tree/master .

库

很多的类库提供了Clojure Proper所没有提供的一些功能， 我们在前面的例子里面已经讨论过一些，下面列举一下没有提到的一些。并且这里有已知的类库的一个列表 http://clojure.org/libraries 。

- clojure.tools.cli - 操作命令行参数并且输出帮助信息
- clojure.data.xml - 以lazy的方式解析XML
- clojure.algo.monads - 有关 monads) 的一些方法
- clojure.java.shell - 提供一些函数和宏来创建子进程并且控制它们的输入/输出
- clojure.stacktrace - 提供函数来简化stacktrace的输出 --- 只输出跟Clojure有关的东西
- clojure.string - 提供操作字符串以及正则表达式的一些方法
- clojure.tools.trace - 提供跟踪所有对某个方法的调用的输出以及返回值的跟踪

下面是个简要的例子要使用 clojure.java.shell 获取当前的工作目录。

```
(use 'clojure.java.shell)
(def directory (sh "pwd"))
```

总结

这篇文章涵盖了很多的背景知识。如果你想学更多有关Clojure的东西， Stuart Halloway 写了本很不错的书：" Programming Clojure "。

这篇文章主要关注的是Clojure 1.0的特性， 并且会被社区成员不时的更新的。如果要了解Clojure 1.1以及更新版本的新特性，可以看看这里：
http://www.fogus.me/static/preso/clj1.1+/

这里有一些关键的问题，你可以问问你自己来看看你到底要不要学习Clojure：

- 你是想要找一种方式使得并发编程更简单么？
- 你确定能够接受一种和面向对象完全不同的编程方式：函数式编程么？
- 能运行在JVM上面， 并且可以调用java的类库，利用java的可移植性对你写的程序重要么？
- 和静态类型比起来你更喜欢动态类型么？
- 你觉得Lisp的简洁的，一致的语法动人么？

如果对于上面某些问题的回答是肯定的，那么你应该考虑尝试下Clojure。

参考

- 我的Clojure网站- http://www.ociweb.com/mark/clojure/
- 我的STM网站- http://www.ociweb.com/mark/stm/
- Clojure官网 - http://clojure.org/
- Clojure API - http://clojure.org/api/
- clj-doc - http://clj-doc.s3.amazonaws.com/tmp/doc-1116/
- Clojure类图 - http://github.com/Chouser/clojure-classes/tree/master/graph-w-legend.png
- clojure-contrib文档 - http://code.google.com/p/clojure-contrib/wiki/OverviewOfContrib/
- Wikibooks：Clojure Programming - http://en.wikibooks.org/wiki/Clojure_Programming
- Wikibooks：Learning Clojure - http://en.wikibooks.org/wiki/Learning_Clojure
- Wikibooks：Clojure API Examples - http://en.wikibooks.org/wiki/Clojure_Programming/Examples/API_Examples
- Project Euler Clojure code - http://clojure-euler.wikispaces.com/
- Clojure Snake Game - http://www.ociweb.com/mark/programming/ClojureSnake.html